International Journal of Information Security
https://doi.org/10.1007/s10207-024-00821-3

REGULAR CONTRIBUTION

A context-aware on-board intrusion detection system for smart vehicles

Davide Micale[1] · Ilaria Matteucci[2] · Florian Fenzl[3] · Roland Rieke[3] · Giuseppe Patanè[1]

Abstract
Modern vehicles are becoming more appealing to potential intruders due to two primary reasons. Firstly, they are now equipped with various connectivity features like WiFi, Bluetooth, and cellular connections, e.g., LTE and 5G, which expose them to external networks. Secondly, the growing complexity of on-board software increases the potential attack surface. In this article, we introduce CAHOOTv2, a context-sensitive intrusion detection system (IDS), aiming at enhancing the vehicle's security and protect against potential intrusions. CAHOOTv2 leverages the vehicle's sensors data, such as the amount of steering, the acceleration and brake inputs, to analyze driver habits and collect environmental information. To demonstrate the validity of the algorithm, we collected driving data from both an artificial intelligence (AI) and 39 humans. We include the AI driver to demonstrate that CAHOOTv2 is able to detect intrusions when the driver is both a human or an AI. The dataset is obtained using a modified version of the MetaDrive simulator, taking into account the presence of an intruder capable of performing the following types of intrusions: denial of service, replay, spoofing, additive and selective attacks. The sensors present in the vehicle are a numerical representation of the environment. The amount of steering, the acceleration and brake inputs given by the driver are based on the environmental situation. The intruder's input often contradicts the driver's wishes. CAHOOTv2 uses vehicle sensors to detect this contradiction. We perform several experiments that show the benefits of hyperparameter optimization. Indeed, we use a hyperparameter tuning paradigm to increase detection accuracy combining randomized and exhaustive search of hyperparameters. As a concluding remark, the results of CAHOOTv2 show great promise in detecting intrusions effectively.

Keywords Automotive · Intrusion detection system · Context-aware · Machine learning · Smart vehicle

1 Introduction

In 2021, there were around 236 million connected vehicles worldwide. This number is projected to rise significantly by 2035, reaching approximately 863 million connected vehicles [36]. Inside these vehicles, the electrical control units (ECUs) play a crucial role in providing various functions for the car [11]. These ECUs are interconnected through different buses, such as controller area network (CAN), CAN-FD, FlexRay, and automotive Ethernet. Various partitions of these buses are interconnected as well via gateways, forming a complex network within the vehicle.

Modern vehicles are not only connected to each other but also to the roadside infrastructure units through V2X communication. This enables each vehicle, and consequently the driver, to receive information about the surrounding environment that may influence driver's decisions. In addition, many newer vehicles are connected to the carmakers' servers via LTE or 5G. Carmakers collect information about 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

✉ Ilaria Matteucci
ilaria.matteucci@iit.cnr.it

Davide Micale
davide.micale@parksmart.it

Florian Fenzl
florian.fenzl@sit.fraunhofer.de

Roland Rieke
roland.rieke@gmx.de

Giuseppe Patanè
giuseppe.patane@parksmart.it

[1] Park Smart SRL, Catania, Italy

[2] IIT-CNR, Pisa, Italy

[3] Fraunhofer SIT, Darmstadt, Germany

Published online: 28 March 2024

electrical vehicles. In particular, carmakers can offer access to the sensors' data to third parties.

As a result, vehicles are increasingly resembling a computer on wheels: the on-board software is becoming even more complex. Today's vehicles contain hundreds of millions lines of code [7]. However, fully autonomous vehicles that do not require any human intervention will contain up to a billion lines of code [7]. For autonomous functionality, cars need numerous sensors to keep track of the environment and the vehicle status [50]. The sensors' data can be accessed internally through the CAN bus protocol or from the external using an OBD-II diagnostic port [43]. In case of an autonomous car, sensors' data are processed by programmable components, such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs) [8], to improve the driver's experience.

In summary, the increasing connectivity of vehicles combined with the increasing complexity of vehicle software can facilitate vehicle intrusions. The *attack surface* in this environment is the sum of the possible vulnerabilities that an attacker can exploit in order to unauthorized extract, inject or manipulate data. The attack surface can formally be defined in terms of the actions and resources of a system that an external user can attack. The work of Manadhata and Wing [25] describes in detail how a system's attack surface can be defined and measured. Keeping the attack surface as small as possible is a fundamental security measure.

Several examples of vehicular attacks have been reported in the literature over the last decade. In 2016, a vulnerability in the web browser of Tesla vehicles allowed an intruder to remotely send messages in the CAN bus [5]. For instance, Weinmann and Schmotzle found a vulnerability in a software component of Tesla that allowed them to unlock the doors and trunk, change seat positions and change both steering and acceleration modes [47]. In general, an intruder may exploit local or remote vulnerabilities of a car to gain some digital access to it, either locally or remotely. She may then modify the behavior of a target vehicle by sending customized CAN frames that trigger a specific functionality on a receiving ECU. Also, using a privilege escalation exploit, it is possible to use the compromised vehicle to compromise surrounding vehicles.

In 2022, the EUROPOL has arrested 31 criminals that were selling a tool, marketed as a diagnostic tool, to replace the original software of the vehicle. The software replacing allowed the criminals to steal keyless cars from two French carmakers without using the original keys [9].

The study of how to protect the CAN buses from in-vehicle vulnerabilities is extremely important. The standard ISO/IEC 27039:2015 [14] and the regulation number 155 of the UNECE (UNECE R155), delivered in 2021, of the United Nations [32] prescribed the use of intrusion detection and prevention systems (IDPS) to monitor the vehicles from intrusions. Under the IDPS umbrella, an intrusion detection system (IDS) merely reports an intrusion alert, while an intrusion prevention system (IPS) alerts and prevents the intrusions. In particular, vehicular context-aware IDSs use the semantic of the messages to detect intrusions.

In the last years, there has been an increase in interest in the metaverse, a virtual 3D representation that mimics real world [13]. Each user is represented by an avatar through which she can interact with the virtual environment and other users. [6] shows how to implement autonomous vehicles in the metaverse and the opportunities offered by their deployment. First, the AI of autonomous vehicle can be tested in safety. Users can use the autonomous vehicle as robot taxi for free roaming. Also, the surveillance systems minimize the risk of damage to cars by car-sharing customers. Finally, all vehicles may be managed by a single system to increase traffic safety. We argue that vehicles in the metaverse can also be targeted by intruders and can be compromised. Hence, an IDS plays a crucial role in protecting against potential intrusions. Also, the metaverse can be an opportunity to safely test the IDS while intrusions are in progress.

In this article, we present CAHOOTv2, an enhancement of CAHOOT [28], a context-aware IDS that can detect tampering with an in-vehicle message-sequence related to a driver's driving style. Indeed, CAHOOT is the first IDS based also on context information able to detect replay and DoS attack in addition to the spoofing attack.

Contextual information allows CAHOOTv2 to better detect intrusions. For example, if a driver accelerates and a sensor detects an obstacle in front of the vehicle, CAHOOTv2 classifies this behavior as a possible intrusion. The environment context is digitally represented by the sensors' values.

1.1 Challenges

The challenges for CAHOOTv2 are:

- Performance: an IDS should be able to process in real time the messages received from the vehicle's sensors. In addition, the low use of computing power enables its implementation in low-cost devices.
- High detection accuracy: intrusions must be detected highly effectively while keeping false positives low, i.e., mistakenly recognizing a driver input as an intrusion.
- Detection of several attacks: an IDS should recognize effectively several types of intrusions.
- Training and validation of a machine learning model using a dataset with several drivers: to demonstrate effectiveness in detecting intrusions, the IDS should be trained and tested on a dataset containing several drivers.

1.2 State of the art

The complexity of modern vehicles proposes several challenges in various aspects of both environmental and safety. In the following, we briefly recall some literature on vehicle communication and then we focus on the most relevant works in the IDS field.

Fatemidokht et al. [10] investigate the difficulties in routing protocols for vehicular ad hoc networks (VANETs) in intelligent transportation systems (ITS). In particular, they propose VRU, a routing protocol that uses unmanned aerial vehicles (UAVs) to improve data routing and identify hostile vehicles. Their findings suggest that adding UAVs into automotive networks can result in considerable improvements in packet delivery ratio, detection ratio, end-to-end latency, and overhead.

Sharma et al. [42] offer a misbehavior detection technique to defend against interruptions induced by erroneous information exchange in the Internet of Vehicles (IoVs), with a focus on the security aspects of vehicular collaboration. The suggested technique detects fake message generation attacks and Sybil attacks with high accuracy, contributing to the overall dependability of vehicular networks.

Deep reinforcement learning and Q-network are used in [33] to improve pedestrian and autonomous vehicle safety. The suggested approach achieves a greater prediction rate accuracy than existing techniques by leveraging LiDAR sensor data and cloud-based predictions. The combination of intelligent observation and reinforcement learning not only assures safer transportation for people and automobiles, but it also helps to reduce fuel consumption and carbon emissions.

The research in [1] offers a high-performance long short-term memory (LSTM) neural network predictor for predictive modeling of hybrid electric cars. The research looks at real-world driving scenarios and compares two architectures for time series prediction models. The LSTM model surpasses other models in properly forecasting vehicle velocity under various conditions, demonstrating its potential application in real-time automotive controllers.

There are several works about the IDS in the vehicular context. Jeong et al. [15] trains an autoencoder to detect anomalies on the sensor values received by the ECUs. Of the six tested layers for the autoencoder, the binary long short-term memory (BiLSTM) is the most accurate.

Pascale et al. [34] introduce an IDS that uses a Bayesian network to detect malicious messages on the CAN bus. The autonomous AI offered by the driving simulator CARLA is used to generate the dataset for model training.

In RAIDS [16] and [21] the IDSs detect intrusions exploiting the images from the on-vehicle camera and the CAN messages. Each work uses two convolutional neural networks trained to detect spoofing attacks.

Xue et al. [48] introduced an IPS that uses the vehicle dynamics to detect intrusions. In particular, the authors define policies starting from the specifications of the target vehicle, in-vehicle messages and onboard sensors to detect intrusion that could affect the safety of the driver.

The detection of sequence context anomalies can be done using different approaches. Rieke et al. [41] used process mining. Levi et al. [22] and Narayanan et al. [29] proposed work using hidden Markov models. Theissler et al. [45] used a one class support vector machine (OCSVM), while the work of Kang et al. [18] uses neural networks. Marchetti et al. [27] used detection of anomalous patterns in a transition matrix. Taylor et al. [44] and Kalutarage et al. [17] used frequency of appearance of a CAN sequence messages.

Lo et al. [24] developed an hybrid neural network IDS composed by a convolutional neural network (CNN) and a long short-term memory (LSTM).

Rajapaksha et al. [39] propose an IDS that uses gated recurrent unit neural network trained only using benign data over CAN messages. A minimum probability threshold is estimated to detect the intrusion. The authors evaluated the work against several publicly available datasets.

Karopoulos et al. [19] propose a new vehicular IDS taxonomy where each IDS belongs to multiple categories. Also the authors provide a survey of the publicly released datasets, simulation tools and IDSs.

The survey of Grimm et al. [12] focuses on the benefits of the context-aware approach on several security fields and the related work. Al-Jarrah et al. [2] provide a survey of IDSs and categorize them. The authors also note the importance of considering the semantics of data and context to detect anomalies.

Micale et al. [28] introduce CAHOOT, the context-aware IDS that detects intrusions either on AI and human drivings for several attacks types. The algorithm is tested using several machine learning algorithms in a dataset made by five humans on a simulator. Random Forest obtained the best results.

The advantages of CAHOOTv2 with respect to CAHOOT are:

- CAHOOTv2 is trained to detect two variants of spoofing attack.
- CAHOOTv2 improves intrusion detection accuracies with respect to CAHOOT. The machine learning algorithms present parameters that must be set before the training process starts and may influence the generated model. These parameters are called hyperparameters [49]. The process of searching the hyperparameters that improve the performance of the models is called hyperparameter tuning [49]. In CAHOOTv2, we design a paradigm that selects the best hyperparameters to use.

– To validate the performance of the algorithm, we also expanded the dataset by collecting driving data from 39 people.

Also, the advantages of CAHOOTv2 over related context-aware IDSs works are:

– CAHOOTv2 detects replay attacks in addition to DoS, spoofing attacks and variants.
– CAHOOTv2 detects intrusions that target steering, throttle and brake instead of only steering or steering and brake.
– CAHOOTv2 detects intrusions on both AI and human driving.
– CAHOOTv2 has a low use of computational resources.

1.3 Article's structure

The article is structured as follows: the next section presents the attack model. Section 3 describes the CAHOOTv2 algorithm. Section 4 shows the results of our experiments. Section 5 concludes the paper with suggestions of future improvements of the algorithm.

2 Attack model

As attack model we consider an *internal intruder* that can be deployed in: a) ECUs that control the steering wheel, engine and brake b) sensors. The attacker is able to forge and sniff messages and performs the following attacks:

– *DoS* attack: the intruder is able to deny driver input by generating CAN frames in which payload values for steering, throttle and brakes are set to zero.
– *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 activate an ECU functionality. We also consider two spoofing attack variants presented in [16]:
  – *Additive* attack: the intruder uses the current valid CAN frame payload and adds a random value in $\pm[0.2, 0.9]$ to simulate an abrupt steering, acceleration or brake.
  – *Selective* attack: the intruder introduces a CAN frame that contradicts the driver's will. The intruder uses the current valid CAN frame payload and flips the sign if the payload absolute value is greater than 0.3 or adds a random value in $\pm[0.5, 1]$.

3 CAHOOTv2 algorithm

The CAHOOTv2 algorithm is based on CAHOOT [28] and aims to detect more attacks and increase accuracy over the older one by optimizing hyperparameters.

CAHOOTv2 inherits several characteristics from CAHO OT:

– CAHOOT has the ability to detect intrusions while a car is moving by analyzing the semantic of CAN messages.
– The algorithm CAHOOT leverages machine learning (ML) techniques for the intrusion detection.
– The algorithm can also detect intrusions when the driver and the intruder generate the same CAN message value.
– The driver can be a human or an AI.
– CAHOOT detects three types of intrusions that target steering, throttle and brake.

In general, CAHOOTv2 is an IDS consisting of a machine learning model that is trained with driving sessions performed by a driver and intrusion attempts performed by an intruder. The ML algorithm is shown how the driver accelerates, brakes, and steers differently than the intruder would in the same environmental situation. This situation is represented by the vehicle's sensors values. The ML algorithm we use is random forest with the hyperparameters selected by our paradigm. This paradigm first selects hyperparameter values randomly and then performs an exhaustive search among the hyperparameter values that performed best.

In the following, we show the motivation that lead us to improve CAHOOT by developing a refined version of the algorithm, named CAHOOTv2. We describe paradigm CAHOOTv2, the pseudocodes of the attacks and how we integrate them on the intruder's behavior. Then, we explain the paradigm responsible for improving the accuracy.

3.1 Motivation

The motivation for CAHOOTv2 is:

– We use random forest as machine learning method to obtain an IDS that is highly accurate on detection and really fast, as shown in Sect. 4.
– CAHOOTv2 is trained to detect different types of attacks. In particular, we trained CAHOOTv2 to detect the replay attack that is the most difficult to detect: the intruder mimics the behavior of a legit driver to go unnoticed in the eyes of the IDS.
– To demonstrate the validity of the CAHOOTv2 algorithm, we use a simulator to simulate several driving situations and roads. Also, the simulator allows to easily collect driving data from several human drivers and AI.

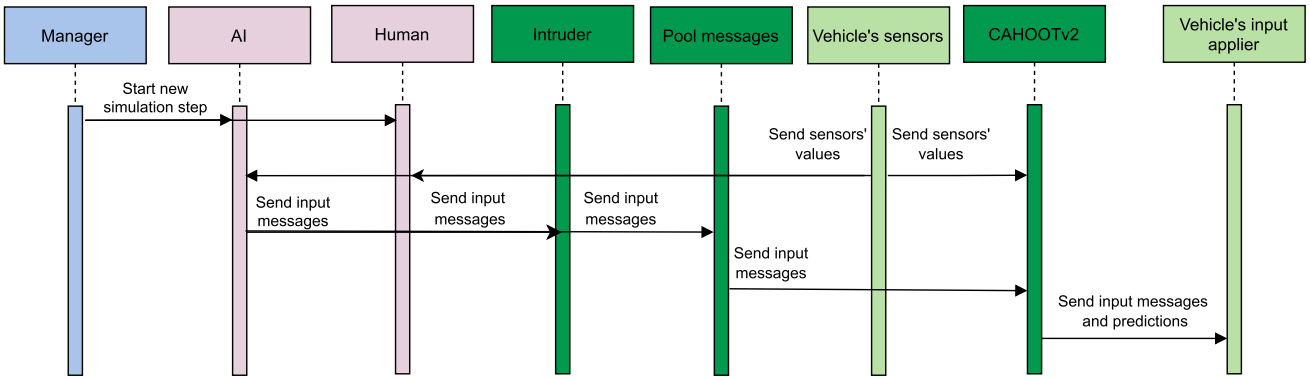

Fig. 1 Inference simulation sequence workflow of the vehicle

3.2 MetaDrive

To evaluate CAHOOTv2, we use a modified version of MetaDrive [23]. MetaDrive is a driving simulator written in Python capable of procedurally generating infinite driving scenarios. Also, the simulator provides a pre-trained AI. We add a logging system in order to collect the values of simulated sensors' vehicle. Also, we introduce an intruder into the MetaDrive simulation workflow (Fig. 1) that sends malicious driving inputs. The in-vehicle communication is represented by a set of messages of two Python lists: the steering messages and the throttle/brake messages sent by both the intruder and the legit driver.

For each step of the intrusion detection workflow (Fig. 1):

- Within the simulator, the Manager starts a new simulation step requesting new vehicle's inputs.
- The legit driver sends driving inputs while an intruder forges fake ones. When the driver is the AI, an AI component gets the values from the vehicle's sensors component and decide what inputs send. When the driver is a human, the simulator gets the inputs from a keyboard, a joystick or a steering wheel.
- Messages are sent to the set of messages present in the Pool messages component. These messages are read by CAHOOTv2.
- The CAHOOTv2 algorithm distinguishes forged messages from the legit ones.
- The component responsible for the steering and the throttle/brake receives the steering wheel and the throttle/brake messages and runs them to the simulated vehicle.
- The simulator clears the set of messages to be able to fill it again in the next simulation step.

Keep note that in the detection phase CAHOOTv2 does not need both legit and forged messages. If the intruder does not forge messages, CAHOOTv2 receives only the legit messages and establishes their legitimacy.

In the training phase, CAHOOTv2 collects at each simulation step the messages of the driver and the intruder from the Pool messages (Fig. 2). CAHOOTv2 needs both messages to train the intrusion detection model.

Similar to humans, AI systems control a vehicle by sending commands for steering, throttle, and brakes based on environmental information. While humans rely on their eyes to estimate the distance from nearby obstacles, AI employs LIDAR and/or vehicle cameras for this purpose. Both humans and AI respond to environmental conditions based on prior experiences, resulting in specific combinations of steering, throttle, and brake inputs. The overarching objective for both human and AI drivers is, to reach a destination, utilizing the vehicle's sensors to gather environmental data and input from the driver. CAHOOTv2 utilizes training data to understand how humans and AI react to environmental information and is also trained to handle responses to intruders. Because the intruder performs actions that are often inconsistent with those that a driver would perform based on the environment, the intruder's actions are detected by CAHOOTv2.

In the following, we describe how the intruder forges the messages inside the simulator and how CAHOOTv2 exploits the messages of drivers and intruders to generate a model that detects intrusions.

3.3 Intruder's behavior

In CAHOOTv2, the intruder frequently changes the attacks randomly choosing among the five described in Sect. 2. The duration of attacks is randomly chosen in an arbitrary interval of steps duration.

Listing 1 and Listing 2 describe the behavior of the intruder. In particular, Listing 1 describes the algorithm *prepare_attack* that plans the duration of each vehicle intrusion, while Listing 2 delves the algorithm *launch_attack*.

Figure 3 provides an overview of the intruder's entire behavioral process.

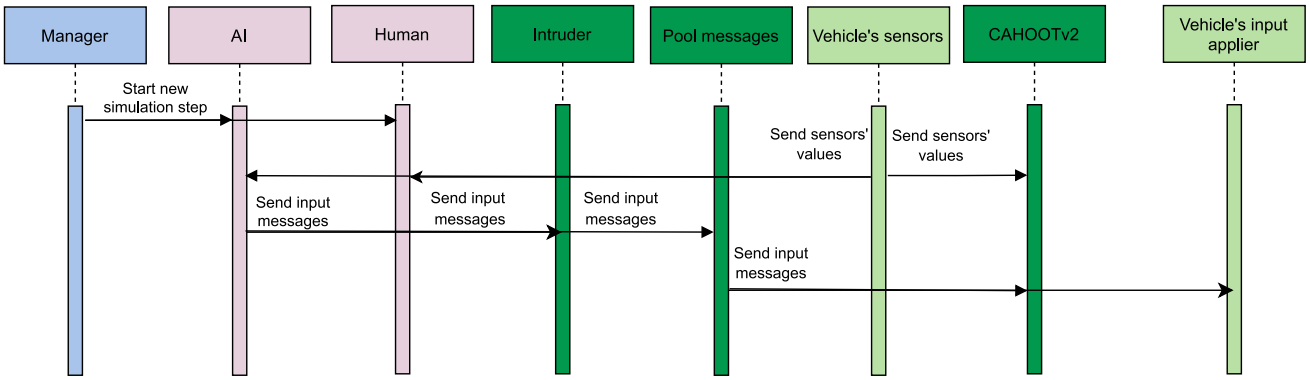

Fig. 2 Training simulation sequence workflow of the vehicle

Listing 1 Prepare Attack

```
1  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)
2    should_attack_change ← stop_attack_time <= Current
       timestamp
3
4    if should_attack_change
5      num_slots ← Select an integer number between
         min_duration and max_duration
6      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)
```

The algorithm *prepare_attack* determines whether the current attack should be continued or altered, i.e., the algorithm compares the actual time with the moment at which the intrusion must be halted (line 2). If the attack gets stopped and replaced with a new attack type, the algorithm determines the duration of the new attack as time slots. The number of slots between the minimum and maximum is randomly chosen by the algorithm (line 5). As a result, the attack will terminate at the sum of the actual time and the product of the number of slots and the duration of each slot (lines 6). To mimic several attacks in a single driving session, the attacks are periodically halted and replaced with new ones.

Whether or not the attack should be changed, the function *launch_attack* is invoked (line 10). The function receives in input the $steering_{legit}$ and $throttle\_brake_{legit}$ to eventually perform an additive or selective attack. When the function ends the execution, it returns the new forged messages along with the current kind of attack, the index of the following messages that the replay attack must repeat, i.e., *index_history*, and the last pair of fabricated messages that the spoofing attack must reproduce, i.e., *prev_steering* and *prev_throttle_brake*. Then, the human/AI steering and throttle_brake inputs are logged in the arrays *steering_history* and *throttle_brake_history* (lines 12 and 13). These arrays may eventually be employed in the replay attack. The attack inputs are never added to the arrays since the replay attack's purpose is to replicate human/AI inputs, hence the attack should only repeat human/AI inputs.

Finally, the algorithm returns the intruder's steering and throttle_brake values, the kind of attack now in execution, the time when the attack will be halted, and the steering and throttle_brake history values, the *index_history*, *prev_steering* and *prev_throttle_brake* (line 15).

Listing 2 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"
```

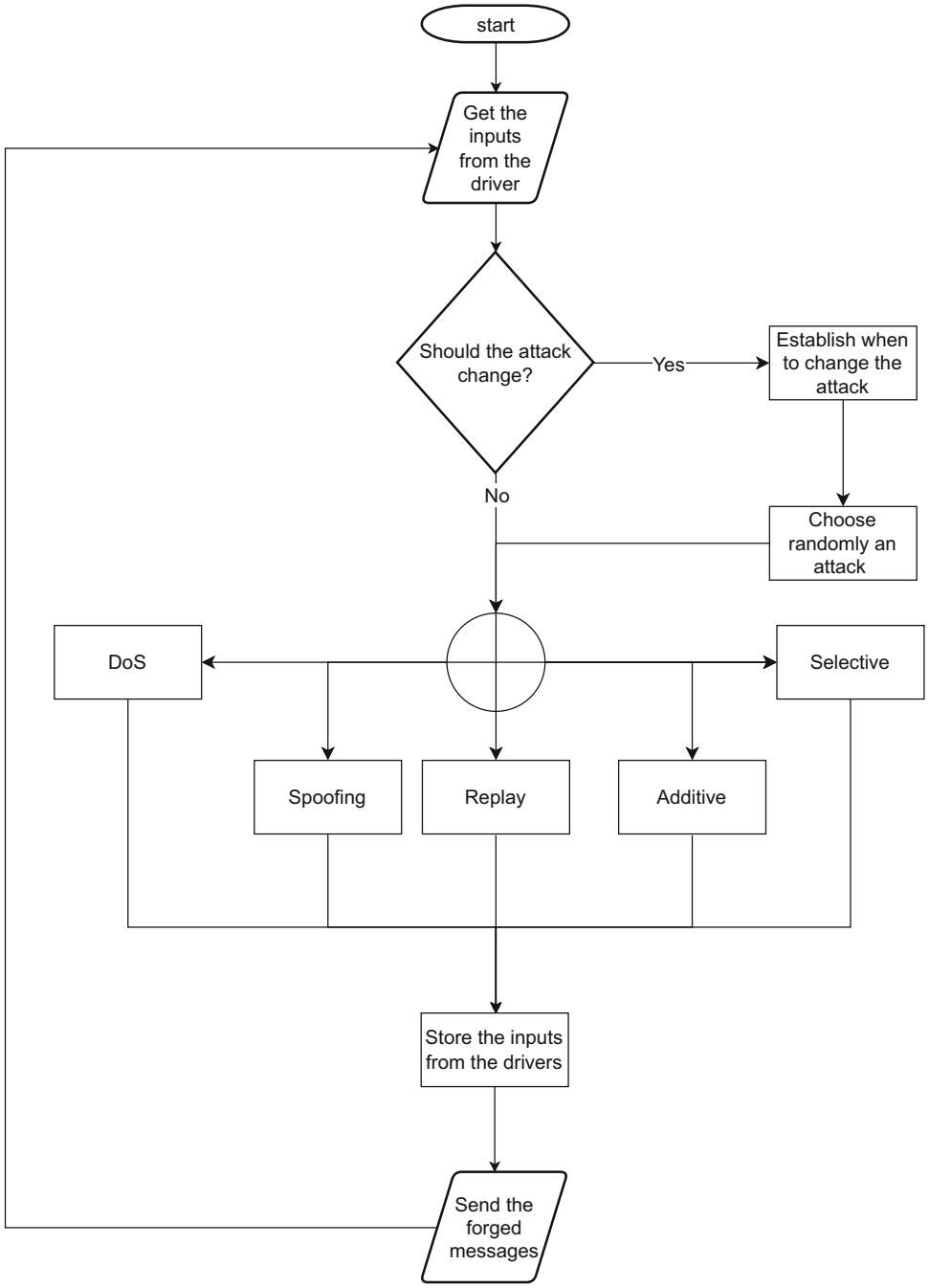

Fig. 3 Inference simulation sequence workflow of the vehicle

```
        (steering, throttle_brake) ←
            spoofing_attack(bootstrap,
            prev_steering, prev_throttle_brake)

        prev_steering ← steering
        prev_throttle_brake ← throttle_brake
    if current_attack = "Replay"
        (steering, throttle_brake, index_history) ←
            replay_attack(bootstrap,
            steering_history, throttle_brake_history,
            index_history)
    if current_attack = "Additive"
        (steering, throttle_brake) ←
            additive_attack(steering_legit,
            throttle_brake_legit)
    if current_attack = "Selective"
        (steering, throttle_brake) ←
            selective_attack(steering_legit,
            throttle_brake_legit)

    return (steering, throttle_brake, current_attack
        , index_history, prev_steering,
        prev_throttle_brake)
```

Listing 2 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 attacks 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* when the attack is active (line 2) and *True* in case the attack is not active (line 4). In case an attack is not in progress, the attack type is randomly chosen between DoS, spoofing, replay, additive and selective (line 6). Based on the current attack type, an attack is launched (lines 8 to 20). In case of spoofing attack, the new steering and throttle_brake messages forged by the intruder are stored, respectively, in *prev_steering* and *prev_throttle_brake* (lines 13 and 14).

Finally, the *launch_attack* function returns the attack's steering and throttle_brake values, the current type of attack, the *index_history* value selected by the replay attack function the last time it is launched, and the previous steering and throttle_brake values used by the spoofing attack (line 22).

3.3.1 Description of considered attacks

The *dos_attack* function sets *steering* and *throttle_brake* functions to 0.

The *spoofing_attack* function sets the steering and the throttle_brake variables with random values (Listing 3). In case the attack is not yet started, the algorithm randomly chooses values between the minimum and maximum valid values (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 3 Spoofing Attack

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

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 4). 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 selects from which position of the array starts to repeat the previous inputs through the setting of *index_history* variable (line 8).

Whether or not the attack has already been launched, the algorithm uses 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 the next steering and throttle_brake pair 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 inputs of the driver. Finally, *replay_attack* function returns the inputs and the new *index_history* (line 15).

Listing 4 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
10         steering ← steering_history[index_history]
11         throttle_brake ← throttle_brake_history
               [index_history]
12
13         index_history ← index_history + 1
14
15     return (steering, throttle_brake, index_history)
```

Additive and selective attacks add a random value to $steering_{legit}$ and $throttle\_brake_{legit}$. The sum operation, made by the additive and selective attacks, may lead to a value that is not valid. The function *limit_value* (Listing 5) ensures that values greater than the *upper_bound*, i.e., the maximum valid value, are changed to *upper_bound* (lines 5 and 6) and values lower than the *lower_bound*, i.e., the minimum valid value, are changed to *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 set to 1 and -1, respectively.

Listing 5 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 sets the *steering* and *throttle_brake* with random values (Listing 6). First, two values are randomly generated in $\pm[0.2, 0.9]$ (lines 2 and 3). Then, these values are added to $steering_{legit}$ and $throttle\_brake_{legit}$. 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 6 Additive Attack

```
1  function additive_attack(steering_legit,
       throttle_brake_legit)
2      random_value_1 ← random value in ±[0.2, 0.9]
3      random_value_2 ← random value in ±[0.2, 0.9]
4
5      steering ← steering_legit + random_value_1
6      throttle_brake ← throttle_brake_legit +
           random_value_2
7
8      steering_limited ← limit_value(steering)
9      throttle_brake_limited ←
           limit_value(throttle_brake)
10
11     return (steering_limited, throttle_brake_limited)
```

The *selective_attack* function creates a *steering* and *throttle_brake* pair based on the value of the legit ones (Listing 7). In case, $steering_{legit}$ is in $\pm[0, 0.3]$, a random value in $\pm[0.5, 1]$ is added to $steering_{legit}$ (lines from 2 to 4). In case $steering_{legit}$ 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 7 Selective Attack

```
1  function selective_attack(steering_legit,
       throttle_brake_legit)
2      if steering_legit in ±[0, 0.3]
3          random_value ← random value in ±[0.5, 1]
4          steering ← steering_legit + random_value
5      else
6          steering ← -steering_legit
7
8      if throttle_brake_legit in ±[0, 0.3]
9          random_value ← random value in ±[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)
```

3.3.2 Instances extraction paradigm

CAHOOTv2 requires a training dataset that contains both legit and forged messages. We label them as follows: $steering_{legit}$, $steering_{forged}$, $throttle\_brake_{legit}$ and $throttle\_brake_{forged}$, alongside with the sensors' values (Table 1).

The *instances_extraction* paradigm extracts the instances of the dataset to generate the final dataset. The new dataset contains messages organized in pairs, each one is labeled as *T* when it contains only legit messages, otherwise it is labeled as *F* (Table 2). With the organization in pairs, CAHOOTv2 is able to detect intrusions when an intruder sends the same message the driver sent. Let us suppose that the driver is not turning the steering wheel, i.e., $steering_{legit}$ is equal to 0, while the intruder starts a DoS attack, i.e., $steering_{forged}$ is equal to 0 (Table 1, row 3). The paradigm considers both the steering message forged by the intruder and the driver as legit since they are equal. However, based on the $throttle\_brake_{legit}$ and $throttle\_brake_{forged}$ the paradigm raises an alert (Table 2, rows 9 and 10). In case both the driver and the intruder send the same pair of messages (Table 1, row 4), the algorithm inserts in the dataset only an instance labeled with *T* (Table 2, row 11).

We execute the *instances_extraction* function (Listing 8) in the initial dataset and the function returns the dataset $ins_{extracted}$ which contains the final produced dataset.

The algorithm begins by reading each *instance* of the initial dataset *ins* (line 3) and organizing the messages into two arrays. Each array comprises tuples formed of steering or the throttle_brake message and a Boolean value denoting the legitimacy of the message.

The two arrays are employed for arranging all of the instances in the initial dataset in a way in which legit and forged messages can be easily distinguished: legitimate messages are loaded into the arrays (lines 12 and 13), whereas forged messages are added only if they differ from the legit messages (lines from 15 to 18).

The $steering_{legit}$, $steering_{forged}$, $throttle\_brake_{legit}$ and $throttle\_brake_{forged}$ messages are removed from *instance* (line 20).

Table 1 Example of instances before run instances extraction paradigm [28]

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	...

Table 2 Example of instances after run instances extraction paradigm [28]

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

As a result, *instance* now contains the engine runtime as well as the sensor values.

The algorithm generates numerous instances based on *instance*, one for each combination of the steering and throttle_brake messages found in *steering_array* and *throttle_brake_array*, respectively (lines 25 and 26).

Next, each instance is labeled `"T"` if it only contains messages from the driver or `"F"` if it contains at least one message from the intruder (lines from 28 to 31).

Finally, all labeled instances are inserted into the $ins_{extracted}$ dataset (line 33). The algorithm returns the dataset $ins_{extracted}$ after reading all the instances in *ins* (line 35).

Listing 8 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"]
7          throttle_brake_forged ←
               instance["throttle_brake_forged"]
8
9          steering_array ← empty array
10         throttle_brake_array ← empty array
11
12         steering_array ← steering_array ⋃
               (steering_legit, True)
13         throttle_brake_array ← throttle_brake_array
               ⋃ (throttle_brake_legit, True)
14
15         if steering_legit != steering_forged
16             steering_array ← steering_array ⋃
                   (steering_forged, False)
17         if throttle_brake_legit != throttle_brake_forged
18             throttle_brake_array ←
                   throttle_brake_array ⋃
                   (throttle_brake_forged, False)
19
20         remove from instance the columns
               "steering_legit", "steering_forged",
               "throttle_brake_legit", "throttle_brake_forged"
21
22
23         for each (steering, is_steering_legit) in
               steering_array
24             for each (throttle_brake,
                   is_throttle_brake_legit) in
                   throttle_brake_array
25                 instance["steering"] ← steering
26                 instance["throttle_brake"] ←
                       throttle_brake
27
28                 if is_steering_legit == True and
                       is_throttle_brake_legit == True
29                     instance["label"] ← "T"
30                 else
31                     instance["label"] ← "F"
32
33                 ins_extracted ← ins_extracted ⋃ instance
34
35     return ins_extracted
```

3.4 Model generation

The model generation paradigm uses the instances extraction paradigm to create the training and testing datasets (Listing 9).

More specifically, after randomly splitting the dataset into a training set and a testing set (line 2), the instances for training and testing are retrieved (lines 3 and 4).

We perform the extraction paradigm independently on the training and testing sets to ensure that all combinations of steering and throttle_brake messages from the same initial instance are not disseminated between training and testing set but remain in the same set. The existence of extracted instances of the same original one in both training and testing sets causes data leakage [20]. Data leaking occurs when information from the training set appears unexpectedly in the testing set.

Then, a feature selection (FS) paradigm ranks all features using the *gain ratio* [38] (GR) approach (line 6). Based on the ranks, the best features are selected: the features with rank equal to zero are removed (line 7). Then, the worst features are removed (lines 9 and 10) from both the train and test sets.

Next, the *hyperparameters_tuning* function is called (line 12). Then, 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 9 Model Generation

```
1  function generate_model(ins_labelled, num_iterations,
       cross_validation, params_dist_random_search)
2    (ins_train, ins_test) ← split randomly the instances as training
       and testing sets from ins_labelled
3    ins_extracted_train ← generate_dataset(ins_train)
4    ins_extracted_test ← generate_dataset(ins_test)
5
6    ranking ← GR(instances)
7    features_>0 ← discard features with rank = 0 from ranking
8
9    ins_bf_train ← ins_extracted_train with features features_>0
10   ins_bf_test ← ins_extracted_test with features features_>0
11
12   params_best ← hyperparameters_tuning(ins_bf_train,
       ins_bf_test, num_iterations, cross_validation,
       params_dist_random_search)
13
14   rf ← initialize a Random Forest using params_best
15
16   model ← train rf using ins_bf_train
17   return model
```

3.5 Hyperparameter tuning paradigm

While the parameters of a model are learned from the dataset in the training phase through the machine learning technique, the hyperparameters should be set manually by the data scientist before starting the training phase. In most cases, the default hyperparameters present in the ML frameworks work well. However, the hyperparameters can be tuned to find a model that performs better [37]. In Random Forest, the ML algorithm used in CAHOOTv2, the hyperparameter types are about the structure of each tree present in the forest, the structure of the forest and the randomness. In the following, we introduce some of the hyperparameters that can be fine-tuned, e.g., the number of trees and the maximum number of features. A random forest model is made of trees. The number of trees is one of the factors that influence the complexity of the model. As the number of trees in the model increases, the generalization error of the model will decrease until it tends toward the limit, i.e., the excessive increase of trees will not reduce the accuracy caused by the overfitting [4]. However, too many trees will increase the computational cost. Each tree is made of a series of nested if-then-else statements where a feature of the data is tested, e.g., the vehicle speed is greater than 30 km/h. Each leaf node corresponds to a prediction. In our study case, the prediction is the presence or absence of intrusion. When growing a tree, the feature selected for each node is selected from a random drawn subset of features. A low maximum number of features of each split lead to less correlated trees. A forest of low correlated trees is more stable than the one with high correlated trees[37]. However, if the maximum number of features is set too low, it may result in the construction of trees that primarily rely on irrelevant features. This can consequently lead to trees with low accuracy.

Listing 10 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, it tries 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. To speed up the computation, we apply the first phase only to the train set. There is only a minority of data in the test set, so discarding the test set has a limited impact on the search for the best hyperparameters.

In the following, we explain in detail the first and the second phases. In the inputs of *hyperparameters_tuning* a

bi-dimensional array $params\_dist_{random\_search}$ is present 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 $params\_accuracies_{random\_search}$ containing 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.

A combination of hyperparameters is randomly generated (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 [26] is calculated on the average accuracies of the groups (line 20). The hyperparameters that have 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 sets 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 value currently 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 10 Hyperparameters Tuning Paradigm

```
1  function hyperparameters_tuning(ins_bf_train, ins_bf_test, num_iterations, cross_validation, params_dist_random_search)
2      params_name ← get the names of parameters in params_dist_random_search
3
4      params_accuracies_random_search ← empty array
5      for num_iterations:
6          params ← Empty array
7          for each param_name in params_name:
8              params[param_name] ← choose uniformly random a hyperparameter in params_dist_random_search[param_name]
9          params_accuracy["params"] ← params
10
11         rf ← initialize a Random Forest with params as hyperparameters
12         params_accuracy["accuracy"] ← train rf using cross_validation-fold cross validation applied to ins_bf_train
13
14         params_accuracies_random_search ← append params_accuracy in params_accuracies_random_search
15
16     params_dist_exhaustive_search ← empty array
17     for each param_name in params_name:
18         grouped_accuracies ← group params_accuracies_random_search by param_name and calculate the average accuracy of each group
19
20         third_quartile ← calculate the third quartile on grouped_accuracies["accuracy"]
21
22         params_accuracies_best_subset ← get the elements in grouped_accuracies on which grouped_accuracies["accuracy"] ≥ third_quartile
23         params_dist_exhaustive_search[param_name] ← params_accuracies_best_subset["params"]
24
25     ins_bf ← ins_bf_train ∪ ins_bf_test
26
27     params_best ← None
28     accuracy_best ← 0
29     for each params combination in params_dist_exhaustive_search:
30         rf ← initialize a Random Forest with params as hyperparameters
31         accuracy ← train rf using cross_validation-fold cross validation applied to ins_bf
32         if accuracy > accuracy_best:
33             params_best ← params
34             accuracy_best ← accuracy
35
36     return params_best
```

4 CAHOOTv2 evaluation

Hereafter, we describe how we evaluate the CAHOOTv2 algorithm on a dataset we generated by the MetaDrive simulation workflow depicted in Sect. 3.2.

Table 3 Features description [28]

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

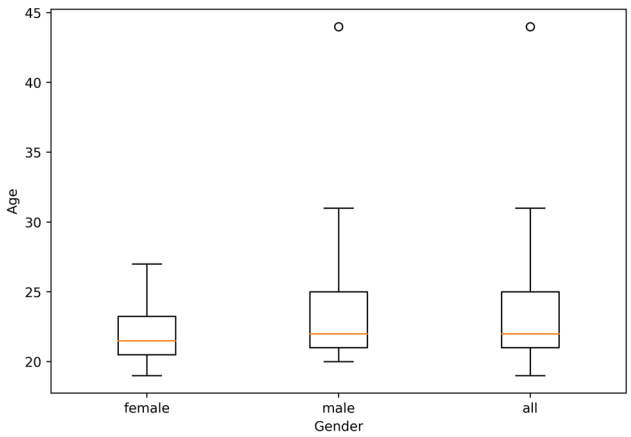

Fig. 4 Boxplots of genders

4.1 Dataset

The dataset is generated using the driving simulator MetaDrive. Table 3 shows the features present in the dataset generated by the MetaDrive simulator.

We aim to obtain an IDS able to work with either humans and AI. Therefore, we decided to collect a dataset that contains data made by an AI and 39 people. In particular, one person uses a keyboard while the remaining 38 use a Thrustmaster TMX [46]. Regarding the gender of the drivers, four drivers are females, while the remaining 35 are males.

Figure 4 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.

AI driving data are collected using the pre-trained AI present in MetaDrive to maintain the consistency of the same simulated vehicle in use between the AI and human drivers.

4.2 Machine learning algorithms

The CAHOOTv2 paradigm is implemented by using python-weka-wrapper3 [40] for the feature selection algorithm gain ratio and scikit-learn [35] that efficiently implements random forest (RF) [4].

Models generated using the 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 [28], hereafter called α. This dataset contains data made by the MetaDrive's AI and 5 people using a Thrustmaster TMX. In the second experiment, we run CAHOOT and CAHOOTv2 on the dataset presented in the previous section, hereafter called β.

4.3 Experiments setup

To evaluate CAHOOTv2, we use several metrics, such as: *Accuracy*, *Precision* and *Recall*.

Accuracy represents how often the model is making a correct prediction.

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

where

– TP (true positive) is the number of instances where at least one sensor's value is forged that are correctly predicted.

Table 4 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"]

– TN (true negative) is the number of instances where all the sensors' values are legit that are correctly predicted.
– FP (false positive) is the number of instances where all the sensors' values are legit but incorrectly predicted.
– FN (false negative) is the number of instances where at least one sensor's value is forged but incorrectly predicted.

Precision measures the ability of the classifier not to predict as forged an instance that is legit. It is calculated as follows:

$$\text{Precision} = \text{TP}/(\text{TP}+\text{FP}) \tag{2}$$

Recall measures the ability of the classifier to find all forged instances. It is calculated as follows:

$$\text{Recall} = \text{TP}/(\text{TP}+\text{FN}) \tag{3}$$

The dataset is randomly split in a training set that contains 85% of instances and a test set that contains the remaining 15%.

The intruder sends forged *steering* and *throttle_brake* messages while the driver is driving the simulated vehicle. Also, multiple attacks on each driving session are simulated through the setting of maximum and minimum duration of an attack, respectively, to 2 and 1 slots.

Table 4 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*.

Table 5 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

4.4 Evaluation of *hyperparameters_tuning*

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

Table 5 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 6 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 (%)
96.0	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 MetaDrive's AI driver					
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

Table 6 shows that the search of hyperparameters increases 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., false negatives increased slightly but false positives decreased.

To better understand on which circumstances the customized hyperparameters perform best, we calculated the accuracy grouped by entity, i.e., human or the MetaDrive's 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 MetaDrive's 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%.

4.5 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. The experiment is conducted on the dataset β.

Table 7 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 55.35% of the entire feature set, in CAHOOT *steering* and *throttle_brake* together rep-

Table 7 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

Table 8 Accuracy, precision and recall comparison on β between CAHOOTv2 and CAHOOTv2 with default hyperparameters

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 MetaDrive's AI driver					
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

resent 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 fifth most important features, in CAHOOT they are only the ninth and the eighth most important features. In CAHOOT, the fourth and fifth most important features are *energy_consumption* and *last_position_x*.

In this case, Tables 8 and 9 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. As stated in Sect. 4.3, precision and recall are metrics that measure the disproportion between the true positives with respect to the false positives and the true positives with respect to the false negatives, respectively. High precision and high recall mean a low number of false positives and false negatives, respectively. Table 8 shows that CAHOOTv2 has only 0.6% difference between precision and recall. Hence, the false positives are slightly higher than the false negatives.

The best hyperparameters allowed to increase the accuracy in each test category and also reduce the differences between the precision and recall.

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 MetaDrive's AI instances, the model with best hyperparameters has an accuracy slightly 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%.

Tables 8 and 9 show that the model easily detects intrusions on instances where the human is driving the vehicle. On the other hand, it is more difficult to detect intrusions on instances where the MetaDrive's AI drives the vehicle. Humans tend to make gradually driving adjustments, whereas MetaDrive's AI makes continuous and sudden changes. Graduality allows the model to detect more precisely an intrusion in progress for human drivings.

CAHOOTv2 with the default hyperparameters is more accurate than CAHOOT by +0.1% on replay and spoofing attacks. Hence, model training on additive and selective attacks allows CAHOOTv2 to increase accuracy on replay and spoofing attacks. The use of hyperparameters further increase the accuracies up to 0.4% on replay attack and up to 0.3% on spoofing attack. The most important increase is in the replay attack because it is the most difficult one to detect. However, CAHOOT is 0.3% more accurate than CAHOOTv2 with default hyperparameters. The use of the best hyperparameters allows CAHOOTv2 to thin the gap with CAHOOT with a decrease of accuracy of only 0.1%. Also, keep note that precision and recall are more balanced in CAHOOTv2 with the best hyperparameters.

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 9 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

The easiest attack to detect in CAHOOT is the spoofing attack, while in CAHOOTv2 with the default hyperparameters is the selective attack. Finally, with the best hyperparameters spoofing and selective attack are equally easy to detect.

Table 10 shows a comparison between the measures of CAHOOTv2 and the main related work on context-aware IDS with respect to the information provided in literature. In fact, this works do not contain experiments on DoS and replay attack. X-CANIDS [15] calls spoofing as masquerade attack and does not provide the accuracy. CARDIAN [34] created spoofing attacks on three scenarios where the car goes straight and: 1) the intruder spoofs the brake sensor, 2) the intruder spoofs the gear, 3) the intruder spoofs the temperature engine. The authors do not provide the accuracy but they provide the TP, TN, FP and FN. Hence, it is possible to calculate the average accuracy. In RAIDS [16] the authors test it on several datasets on additive and selective attacks, but they provided only the worst accuracy obtained in the additive attack and the best accuracy obtained in the selective attack. Also, they call the additive attack as abrupt attack and the selective attack as directed attack. Finally, Kondratiev et al. [21] test two different neural networks trained to detect the intrusions. In Table 10, only the metrics of the best neural network are shown. Authors provide only the recalls for the additive attack while for the selective attack they provide accuracies, precisions and recalls. Also, the authors call the additive attack as random attack.

CAHOOTv2 consistently performs better than the related work. In particular, CAHOOTv2 obtains the same accuracy as CARDIAN in the spoofing attack. However, the precision and the recall are much better in CAHOOTv2 with, respectively, +17.4% and +18.5%. CARDIAN is very good at detecting legitimate messages, but has some difficulty detecting intrusions. CAHOOTv2 has an almost perfect recall with 99.9% while X-CANIDS stops at 91.2%. The precision of CAHOOTv2 differs from X-CANIDS by only 0.5%. CAHOOTv2 is more accurate with respect to the worst accuracy obtained by RAIDS in the additive attack, i.e., +8.2%,

Table 10 Comparison of lowest and highest accuracies on spoofing attack between CAHOOT and the main context-aware IDSs

	Spoofing			Additive			Selective		
	Accuracy	Precision	Recall	Accuracy	Precision	Recall	Accuracy	Precision	Recall
CAHOOTv2	99.6%	99.5%	99.9%	97.7%	99.5%	97.3%	99.6%	99.7%	99.8%
CAHOOT	99.3%	99.1%	99.9%	83.5%	87.1%	91.3%	86.7%	87.9%	95.4%
X-CANIDS [15]	N/A	99.0%	91.2%	N/A	N/A	N/A	N/A	N/A	N/A
CARDIAN [34]	99.6%	82.1%	81.4%	N/A	N/A	N/A	N/A	N/A	N/A
RAIDS [16]	N/A	N/A	N/A	$\geq$89.5%	N/A	N/A	$\leq$99.9%	N/A	N/A
Kondratiev et al. [21]	N/A	N/A	N/A	N/A	N/A	92.5%	73.7%	68.9%	98.3%

Table 11 Comparison of evaluation time between CAHOOT and the main context-aware IDSs

	Response time (μs)	Device	
		Name	Type
CAHOOTv2	29	Jetson Nano	Embedded
CAHOOT	27	Jetson Nano	Embedded
X-CANIDS [15]	8,060	Jetson AGX Xavier	Embedded
CARDIAN [34]	N/A	N/A	Embedded
RAIDS [16]	<400,000	Raspberry Pi 3 Model B+	Embedded
Kondratiev et al. [21]	17,857	Intel Core i5-6300HQ	Laptop

Table 12 Abbreviations

Abbreviation	Description
LTE	Long-term evolution. Fourth-generation cellular connection
5 G	Fifth generation cellular connection
IDS	Intrusion detection system
AI	Artificial intelligence
ECU	Electrical control units
CAN	Controller area network
CAN-FD	Controller area network flexible data-rate
V2X	Vehicle-to-everything
OBD-II	On-board diagnostics II
GPU	Graphics processing unit
FPGA	Field-programmable gate arrays
ISO/IEC	International Organization for Standardization and the International Electrotechnical Commission
UNECE	United Nations Economic Commission for Europe
IDPS	Intrusion detection and prevention systems
IDS	Intrusion detection system
IPS	Intrusion prevention system
DoS	Denial of service
BiLSTM	Binary long short-term memory
OCSVM	One class support vector machine
CNN	Convolutional neural network
LSTM	Long short-term memory
FS	Feature selection
GR	Feature selection
ML	Machine learning
TP	True positive
TN	True negative
FP	False positive
FN	False negative

and it is slightly less accurate with the best accuracy obtained by RAIDS in the selective attack, i.e., only -0.3% accurate. Finally, CAHOOTv2 has a recall higher than Kondratiev et al. [21] in the additive attack, i.e., 4.8%, and has a much higher accuracy and precision in the selective attack, i.e., +25.9% and +30.8%, respectively.

Finally, we evaluate the response time of CAHOOTv2 on intrusion detection (Table 11). We exported the Random Forest model, previously trained by a server, into a NVIDIA Jetson Nano Developer Kit. This device contains a NVIDIA Maxwell GPU, 4 GB of RAM and four cores ARM Cortex-A57 [31]. The evaluation of a single instance requires in average 29μs. We also evaluate CAHOOT on the same device obtaining 27μs, only 2μs faster than CAHOOTv2 (Table 11).

CAHOOTv2 is at least 278 times faster compared to the main related work. In particular, X-CANIDS takes $8{,}060\mu$s despite using the more powerful NVIDIA Jetson AGX Xavier with 8 cores [30]. The use of an autoencoder made with BiLSTM layers is computationally onerous. However, RAIDS and [21] are the slowest methods because they process either the images from the on-vehicle camera and sensors. CARDIAN runs on an embedded device but the authors do not provide the exact device model used for the evaluation and the response time of the IDS.

Due to CAHOOTv2's low use of computational resources, driver data can be analyzed directly in the car and do not need to be sent outside the vehicle, making CAHOOTv2 privacy preserving.

5 Conclusions and future work

The high complexity of newer vehicles increases the attack surfaces on which a vulnerability could be present. An intru-

Table 13 Notations

Notations	Description
Hyperparameters	Parameters that must be set before the training process starts and may influence the generated mode
Hyperparameters tuning	The process of searching the hyperparameters that improve the performance of the models
Context-aware IDS	Use the semantic of the messages to detect intrusions
DoS attack	Steering, throttle and brakes are set to zero
Replay attack	The intruder re-uses valid CAN frames
Spoofing attack	The intruder is able to generate a valid CAN frame
Additive attack	The intruder simulates an abrupt steering, acceleration or brake
Selective attack	The intruder introduces a CAN frame that contradicts the driver's will
MetaDrive	Driving simulator written in Python
Instances extraction paradigm	Extracts the instances of the dataset to generate the final dataset where messages are organized in pairs
Feature Selection paradigm	Ranks and selects the best features

sion while the vehicle is in motion could endanger the lives of the driver and passengers.

In this article, 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 (Tables 12, 13).

Security solutions are strongly linked to safety, especially when considering the automotive domain. CAHOOT and CAHOOTv2 are designed to be an IDS, so malicious events are just identified and no active reactions are implemented to avoid that they may impact the vehicle's safety. While driving a car, events may occur that require exceptional responses from the driver, e.g., a cat suddenly crossing the road forcing an abrupt stop. If not properly trained, an IDS may interpret these events as malicious. CAHOOT and CAHOOTv2 are already trained to identify dangerous situations, e.g., one of the simulated cars performed sudden overtaking or congested traffic that forced the driver to make abrupt decisions.

As future work, we want to further improve our algorithm to not only detect intruders but also identify drivers while maintaining their privacy. Rather than endangering the lives of the driver and passengers in the vehicle, the intruder may be interested to introduce CAN messages to trick the vehicle's driver identification system to impersonate an authorized driver. To prevent this, the intrusion detection component of the algorithm may identify the forged messages and prevent them to reach the driver identification component. As another possible future scope, we will apply CAHOOTv2 to real vehicles. The new vehicles may integrate Android Automotive [3] which is an android operating system for vehicles. Android automotive allows developers to create apps that use the vehicle's sensor data. CAHOOTv2 could be installed in these vehicles as a download from the Play Store. As shown in Sect. 4, CAHOOTv2 can be integrated into low-cost devices that enable affordable IDS production for older vehicles.

CAHOOTv2 can be installed not only for private vehicles, but also for public transportation vehicles and emergency vehicles. An intruder that attacks a bus can endanger more lives than attacking a private vehicle. For this reason, the introduction of an IDS for public transport vehicles is particularly important. Also, an intruder that aims at sabotaging an ambulance may endanger the life of a patient who urgently needs to reach the hospital for treatment. An intruder could prevent a police car from chasing a getaway vehicle.

Finally, CAHOOTv2 is also capable of detecting intrusions in autonomous vehicles. Hence, CAHOOTv2 can be installed in future self-driving cars to detect intrusions in a timely and effective manner.

Another future scope is the introduction of CAHOOTv2 to vehicles of the metaverse. In future, we aim to refine the algorithm to detect which sensor of the car is being attacked. Also, we will try new machine learning techniques that could help to increase the accuracy. In future, we will train the IDS model by collecting data with environmental conditions different from those actually present in MetaDrive: at present, the simulator only simulates sunny days. The model should also be trained with rain, snow and fog. Also, we will consider new driving scenarios, e.g., the driver drives fast because she is late to an appointment.

Funding Open access funding provided by IIT - RENDE within the CRUI-CARE Agreement.

Data availability The data that support this research activity have been collected in a compliant way with respect to ethical and privacy regulation. Data may be disclosed after the participant consent.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Akl, N.A., El Khoury, J., Mansour, C.: Trip-based prediction of hybrid electric vehicles velocity using artificial neural networks. In: 2021 IEEE 3rd International Multidisciplinary Conference on Engineering Technology (IMCET), pp. 60–65 (2021). https://doi.org/10.1109/IMCET53404.2021.9665641
2. Al-Jarrah, O.Y., Maple, C., Dianati, M., Oxtoby, D., Mouzakitis, A.: Intrusion detection systems for intra-vehicle networks: A review. IEEE Access **7**, 21266–21289 (2019)
3. "Android Open Source Project": What is android automotive? (2023). URL:https://source.android.com/docs/automotive/start/what_automotive?hl=en [retrieved: 11, 2023]
4. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001). https://doi.org/10.1023/A:1010933404324
5. "CVE": Cve-2016-9337 (2016). URL:https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9337 [retrieved: 11, 2022]
6. Deveci, M., Pamucar, D., Gokasar, I., Köppen, M., Gupta, B.B.: Personal mobility in metaverse with autonomous vehicles using q-rung orthopair fuzzy sets based opa-rafsi model. IEEE Transactions on Intelligent Transportation Systems pp. 1–10 (2022). https://doi.org/10.1109/TITS.2022.3186294
7. Diess, H.: Levers to unleash value (2020). URL:https://www.volkswagenag.com/presence/investorrelation/publications/presentations/2020/01-januar/January_2020_VWAG_Investor_Roadshow.pdf [retrieved: 11, 2022]
8. Donzellini, G., Garavagno, A.M., Oneto, L.: Microprocessor Systems on FPGA, pp. 439–553. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-87344-8_5
9. "EUROPOL": 31 arrested for stealing cars by hacking keyless tech (2022). URL:https://www.europol.europa.eu/media-press/newsroom/news/31-arrested-for-stealing-cars-hacking-keyless-tech [retrieved: 11, 2022]
10. Fatemidokht, H., Rafsanjani, M.K., Gupta, B.B., Hsu, C.H.: Efficient and secure routing protocol based on artificial intelligence algorithms with UAV-assisted for vehicular ad hoc networks in intelligent transportation systems. IEEE Trans. Intell. Transp. Syst. **22**(7), 4757–4769 (2021). https://doi.org/10.1109/TITS.2020.3041746
11. Gmiden, M., Gmiden, M.H., 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), pp. 176–180 (2016). https://doi.org/10.1109/STA.2016.7952095
12. Grimm, D., Stang, M., Sax, E.: Context-aware security for vehicles and fleets: a survey. IEEE Access **9**, 101,809-101,846 (2021). https://doi.org/10.1109/ACCESS.2021.3097146
13. Hollensen, S., Kotler, P., Opresnik, M.: Metaverse: the new marketing universe. J. Bus. Strategy **44**(3), 119–125 (2023). https://doi.org/10.1108/JBS-01-2022-0014
14. "International Organitation for Standardization: Iso/iec 27039:2015, information technology — security techniques — selection, deployment and operations of intrusion detection and prevention systems (idps) (2015). URL:https://www.iso.org/standard/56889.html [retrieved: 11, 2022]
15. Jeong, S., Lee, S., Lee, H., Kim, H.K.: X-canids: Signal-aware explainable intrusion detection system for controller area network-based in-vehicle network (2023)
16. Jiang, J., Wang, C., Chattopadhyay, S., Zhang, W.: Road Context-Aware Intrusion Detection System for Autonomous Cars. Lecture Notes in Computer Science p. 124–142 (2020). https://doi.org/10.1007/978-3-030-41579-2_8
17. Kalutarage, H.K., Al-Kadri, M.O., Cheah, M., Madzudzo, G.: Context-aware anomaly detector for monitoring cyber attacks on automotive can bus. In: ACM Computer Science in Cars Symposium, CSCS '19. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3359999.3360496
18. Kang, M.J., 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)
19. Karopoulos, G., Kambourakis, G., Chatzoglou, E., Hernández-Ramos, J.L., Kouliaridis, V.: Demystifying in-vehicle intrusion detection systems: A survey of surveys and a meta-taxonomy. Electronics **11**(7) (2022). https://doi.org/10.3390/electronics11071072. https://www.mdpi.com/2079-9292/11/7/1072
20. Kaufman, S., Rosset, S., Perlich, C., Stitelman, O.: Leakage in data mining: formulation, detection, and avoidance. ACM Trans. Knowl. Discov. Data **6**(4), 1–21 (2012). https://doi.org/10.1145/2382577.2382579
21. Kondratiev, V., 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), pp. 1–5 (2021). https://doi.org/10.1109/ITNT52450.2021.9649295
22. Levi, M., Allouche, Y., Kontorovich, A.: Advanced analytics for connected cars cyber security. CoRR arxiv:1711.01939 (2017)
23. Li, Q., Peng, Z., Xue, Z., Zhang, Q., Zhou, B.: Metadrive: Composing diverse driving scenarios for generalizable reinforcement learning. arXiv preprint arXiv:2109.12674 (2021)
24. Lo, W., Alqahtani, H., Thakur, K., Almadhor, A., Chander, S., Kumar, G.: A hybrid deep learning based intrusion detection system using spatial-temporal representation of in-vehicle network traffic. Vehicular Communications **35**, 100,471 (2022). https://doi.org/10.1016/j.vehcom.2022.100471. https://www.sciencedirect.com/science/article/pii/S2214209622000183
25. Manadhata, P., Wing, J.: An attack surface metric. IEEE Trans. Softw. Eng. **37**, 371–386 (2011). https://doi.org/10.1109/TSE.2010.60
26. Mann, P.S.: Introductory Statistics. Wiley, New York (2009)
27. Marchetti, M., Stabili, D.: Anomaly detection of CAN bus messages through analysis of id sequences. In: 2017 IEEE Intelligent Vehicles Symposium (IV), pp. 1577–1583 (2017)
28. Micale, D., Costantino, G., Matteucci, I., Fenzl, F., Rieke, R., Patanè, G.: Cahoot: a context-aware vehicular intrusion detection system. In: 2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 1211–1218 (2022). https://doi.org/10.1109/TrustCom56396.2022.00168
29. Narayanan, S.N., Mittal, S., Joshi, A.: Obd securealert: An anomaly detection system for vehicles. In: IEEE Workshop on Smart Service Systems (SmartSys 2016) (2016)
30. "NVIDIA": NVIDIA® Jetson AGX Xavier™. URL:https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/jetson-agx-xavier/ [retrieved: 07, 2023]
31. "NVIDIA": NVIDIA® Jetson Nano™. URL:https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/jetson-nano/ [retrieved: 07, 2023]
32. "Official Journal of the European Union": Uniform provisions concerning the approval of vehicles with regards to cybersecurity and cybersecurity management system (2021). URL:http://data.europa.eu/eli/reg/2021/387/oj [retrieved: 11, 2022]
33. Vijayakumar, P., Rajkumar, S.C.: Deep reinforcement learning-based pedestrian and independent vehicle safety fortification using intelligent perception. Int. J. Softw. Sci. Comput. Intell. **14**(1), 1–33 (2022). https://doi.org/10.4018/IJSSCI.291712

34. Pascale, F., Adinolfi, E.A., Avagliano, M., Bellacosa, E., Coppola, S., Santonicola, E.: Cardian: A context aware cybersecurity system for real time diagnostic intrusion detection using a probabilistic approach with bayesian network. In: 2022 6th International Conference on System Reliability and Safety (ICSRS), pp. 424–429 (2022). https://doi.org/10.1109/ICSRS56243.2022.10067343
35. 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., Duchesnay, E.: Scikit-learn: Machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
36. Placek, M.: Connected car fleet by region 2021-2035 (2022). URL:https://www.statista.com/statistics/1155517/global-connected-car-fleet-by-market/ [retrieved: 11, 2022]
37. Probst, P., Wright, M., Boulesteix, A.L.: Hyperparameters and tuning strategies for random forest. Wiley Interdiscipl. Rev. Data Mining Knowl. Discov. **9**, e1301 (2019). https://doi.org/10.1002/widm.1301
38. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc., San Francisco (1993)
39. Rajapaksha, S., Kalutarage, H., Al-Kadri, M.O., Madzudzo, G., Petrovski, A.V.: Keep the moving vehicle secure: Context-aware intrusion detection system for in-vehicle can bus security. In: 2022 14th International Conference on Cyber Conflict: Keep Moving! (CyCon), vol. 700, pp. 309–330 (2022). https://doi.org/10.23919/CyCon55549.2022.9811048
40. Reutemann, P.: python-weka-wrapper3 (2020). URL:https://fracpete.github.io/python-weka-wrapper3/index.html [retrieved: 11, 2022]
41. Rieke, R., Seidemann, M., Talla, E.K., Zelle, D., 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, pp. 381–385. IEEE Computer Society (2017)
42. Sharma, R., Sharma, T.P., Sharma, A.K.: Detecting and preventing misbehaving intruders in the internet of vehicles. IJCAC **12**(1), 1–21 (2022). https://doi.org/10.4018/ijcac.295242
43. "Snap-on Incorporated": Global obd vehicle communication software manual (2013). URL:https://www.snapon.com/Files/Diagnostics/UserManuals/GlobalOBDVehicleCommunicationSoftwareManual_EAZ0025B43.pdf [retrieved: 11, 2022]
44. Taylor, A., Japkowicz, N., Leblanc, S.: Frequency-based anomaly detection for the automotive CAN bus. In: 2015 World Congress on Industrial Control Systems Security (WCICSS), pp. 45–49 (2015)
45. 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)
46. "Thrustmaster": TMX Force Feedback. URL:https://www.thrustmaster.com/products/tmx-force-feedback/ [retrieved: 11, 2022]
47. Weinmann, R.P., Schmotzle, B.: TBONE: for public release on 2021-04-28 (2021). URL:https://kunnamon.io/tbone/ [retrieved: 11, 2022]
48. Xue, L., Liu, Y., Li, T., Zhao, K., Li, J., Yu, L., Luo, X., Zhou, Y., Gu, G.: SAID: State-aware defense against injection attacks on in-vehicle network. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 1921–1938. USENIX Association, Boston, MA (2022). https://www.usenix.org/conference/usenixsecurity22/presentation/xue-lei
49. Zhang, A., Lipton, Z.C., Li, M., Smola, A.J.: Dive into Deep Learning (2020). https://d2l.ai [retrieved: 11, 2022]
50. Zheng, B., Liang, H., Zhu, Q., Yu, H., Lin, C.W.: Next generation automotive architecture modeling and exploration for autonomous driving. In: 2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 53–58 (2016). https://doi.org/10.1109/ISVLSI.2016.126