# Integrating Adversary Models and Intrusion Detection Systems for In-Vehicle Networks in CANoe

Camil Jichici, Bogdan Groza, and Pal-Stefan Murvay

Faculty of Automatics and Computers,
Politehnica University of Timisoara, Romania
Email: jichicicamil93@gmail.com, bogdan.groza@aut.upt.ro, pal-stefan.murvay@aut.upt.ro

**Abstract.** In-vehicle buses and the Controller Area Network (CAN) in particular have been shown to be vulnerable to adversarial actions. We embed adversary models and intrusion detection systems (IDS) inside a CANoe based application. Based on real-world CAN traces collected from several vehicles we build attack traces that are subject to intrusion detection algorithms. We also take benefit from existing machine-learning support in MATLAB that is ported via C++ code in CANoe in order to integrate intrusion detection functionality. A unified framework for attacks and intrusion detection has the benefit of providing a testbed for various intrusion detection algorithms. CANoe integration makes the use of these functionalities ready for realistic testing as CANoe is an industry-standard tool in the automotive domain.

**Keywords:** CAN bus · vehicle security · intrusion detection

## 1 Introduction and Related Work

Contemporary vehicles incorporate dozens of Electronic Control Units (ECUs), sensor networks and actuators interconnected through in-vehicle networks such as: Local Interconnect Network (LIN), Controller Area Network (CAN), FlexRay, etc. Access to these in-vehicle networks is mediated by several interfaces, more commonly the On-Board Diagnostic (OBD) port which is used by our work as well (more details in a later section). However, the high complexity and connectivity of modern vehicles leads to cyber security risks which might undermine the privacy of the vehicle and endanger the life of passengers. This was well proved by a strong body of research in [3], [13]. Recent advances regarding autonomous driving, enhanced technologies for infotainment systems and vehicle-to vehicle communications (V2V) will transform vehicles into devices that interact with each other over the Internet and can be remotely controlled. This trend opens even more attack surfaces that were well exploited by recent works [10], [18].

Most of the reported attacks on in-vehicle communication employ the CAN protocol. This is natural as CAN is the most widely used bus in the automotive domain and is often exposed through the diagnostic port. Details on the CAN bus topology, bit rates and the frame format are deferred to Appendix 1.

As vulnerabilities on the CAN bus are easy exploitable by adversaries, the development of intrusion detection systems (IDS) is an immediate necessity in order to quickly detect such attacks. A comprehensive survey on IDSs for in-vehicle buses can be found in [1]. The authors in [1] provide a hierarchical and structured picture of IDS proposed in the literature for passenger cars. There are many relevant proposals, we outline some of them next. Binary distance, i.e., Hamming distance, is proposed in [19]. Their approach includes two stages: a preliminary stage and a detection phase. In the first stage, for each CAN ID the authors calculate the Hamming distances between consecutive CAN frames on 20% of the trace and build message validity ranges bounded by the minimum and maximum distance computed for each ID. The rest of the frames (80% of the recorded trace) are used in the detection phase. An anomaly is detected when the Hamming distance is outside the validity range. In a similar vein, entropy has also been used to detect intrusions in [12] and [15]. Groza et al. proposed an IDSs based on Bloom filtering in [7]. Their detection mechanism filters the transmission frequency and the data field of the frames for each identifier in order to detect replay and modification attacks and takes advantage regarding low consumption of resources which are compulsory in deployment of a real-world IDS.

On the other hand, there are several machine learning and statistic based approaches for in-vehicle networks intrusion detection. Narayanan *et al.* create a Hidden Markov Model based on CAN data collected through the OBD port in order to detect intrusion on CAN bus in [16]. Their model describes vehicle states and possible transitions between them. The intrusion is detected when an unexpected transition occurs. Support vector machine and k-Nearest Neighbor (k-NN) classifier were proposed in [2]. The authors from [2] build a classifier model that is not able to detect replay attacks since they do not use the frequency of the messages in the training phase. In our work, we also use the periodicity of messages to enable identifying replay attacks since, in such attacks, the timestamp of the CAN messages is the single attack indicator. Another proposed approach [9] is the use of deep neural networks. The result of this work is not based on a real-world CAN traffic, instead the authors use CAN traffic generated by a software tool OCTANE [6]. The authors of [9] do not account for message transmission frequency in the training phase which leads to the inability to detect replay attacks. Decisions tree having as inputs entropy-based characteristics extracted from CAN IDs and timestamps were used in [21].

The idea of using CANoe (i.e. an industry-standard tool in the automotive domain), for evaluating security is not new and has been explored by previous works. Some of the first simulation-based attacks on the CAN [8] and FlexRay [17] examine vulnerabilities of simulated in-vehicle networks based on the CAN and FlexRay protocol respectively to spoofing and replay attacks. We complement these ideas by integrating intrusion detection along with the adversarial model in CANoe. Thus, our work aims to provide a more complex framework based on the support from two widely-used tools in the automotive-industry, CANoe and Matlab, in order to simulate adversarial actions and detect them in real-world scenarios.

## 2 Data Collection and Experimental Setup

In this section we first discuss how data collection was performed and then how we use it in CANoe simulation.

## 2.1 Data Extraction from OBD

In order to develop an intrusion detection mechanism based on real-world CAN traces, we first collect data from the CAN bus via the OBD port of several cars. The OBD port aims to collect diagnostic data from all ECUs. Consequently, in most cases, it is connected directly to the main CAN bus of the vehicle. For enhancing the in-vehicle security, the OBD port should be directly connected only to an ECU gateway which then collects the diagnostic information from all other ECUs in order. In such case, only diagnostic messages corresponding to request-response protocol would be visible through the OBD port. However, in order to reduce costs, many vehicles do not have such a gateway ECU and the OBD port is connected to the main CAN bus. For the cars employed in our experiments we determined that in-vehicle traffic is indeed exposed over the OBD port.

As a first step, to enable data collection, we determined if there is any traffic exposed to the CAN pins of the OBD port and what is the employed bit rate. We achieved this with the help of an oscilloscope revealing that CAN traffic is indeed available and that one of the vehicles uses a baud rate of 250 Kbit/s while the other uses 500 Kbit/s. Then, we proceeded to logging the traffic from CAN bus for about 20 minutes with the car stationary and 20 minutes with the car in motion. During this interval, several driver-specific actions were performed, e.g., toggling low beam and long beam, sudden accelerations and brakes, etc. This was done for both the cars that we used, a sedan and an SUV.

Figure 1 depicts our experimental setup based on the Vector VN1630 USB-to-CAN interface, the OBD plug, the CAN cable and an application based on the Vector XL Driver Library running on the laptop. The VN1630 device is a part of the VN1600 family developed by Vector, a provider of solutions for automotive networking development. Support for the VN1630 exists in a number of software tools such as CANoe and CANape along with support for building dedicated applications through the XL Driver Library. The XL Driver Library is an Application Programming Interface (API) compatible with Vector's devices. The library provides access to device functionalities (e.g., message reception and transmission, various configuration settings) and handles interfacing with protocols such as CAN, CAN-FD, LIN, FlexRay, etc. An example of messages intercepted from the vehicle CAN bus (via the OBD port) with a small application using this library is shown in Figure 2. Each message consists of the VN1630 channel number on which the message is received, the timestamp (in nanoseconds), the message identifier, the length (in bytes) of the data field and the actual data field. In our work, we run an application that only receives messages from the vehicles and did not try to inject frames in the car to avoid potential damage to the vehicle. Consequently, the injections will be performed in the CANoe simulation.

## 2.2 CANoe Environment

We integrate the attacker model and intrusion detection capabilities in a CANoe simulation. CANoe is an unified integrated software used for designing, simulating, testing and analyzing real-time communication between ECUs. In the automotive domain, CANoe is the most wide-spread tool used by automotive manufacturers in the development process of in-vehicle networks.

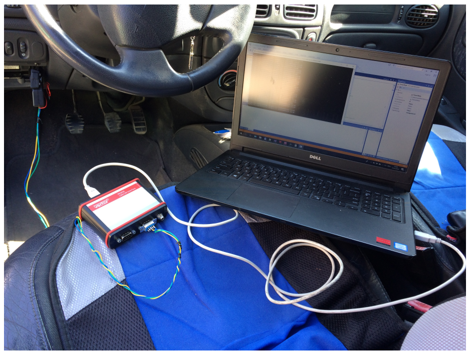
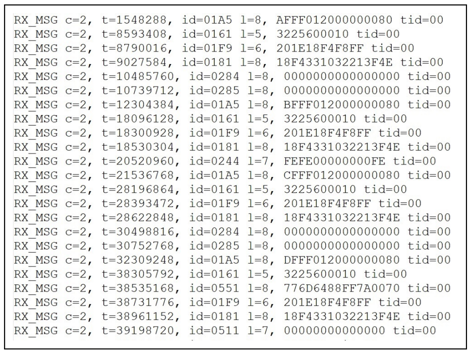


Fig. 1: Setup for data collection inside car



```
RX_MSG c=2, t=1548288, id=01A5 l=8, AFFF012000000080 tid=00
RX_MSG c=2, t=8593408, id=0161 l=5, 3225600010 tid=00
RX_MSG c=2, t=8790016, id=01F9 l=6, 201E18F4F8FF tid=00
RX_MSG c=2, t=9027584, id=0181 l=8, 18F4331032213F4E tid=00
RX_MSG c=2, t=10485760, id=0284 l=8, 0000000000000000 tid=00
RX_MSG c=2, t=10739712, id=0285 l=8, 0000000000000000 tid=00
RX_MSG c=2, t=12304384, id=01A5 l=8, BFFF012000000080 tid=00
RX_MSG c=2, t=18096128, id=0161 l=5, 3225600010 tid=00
RX_MSG c=2, t=18300928, id=01F9 l=6, 201E18F4F8FF tid=00
RX_MSG c=2, t=18530304, id=0181 l=8, 18F4331032213F4E tid=00
RX_MSG c=2, t=20520960, id=0244 l=7, FEFE00000000FE tid=00
RX_MSG c=2, t=21536768, id=01A5 l=8, CFFF012000000080 tid=00
RX_MSG c=2, t=28196864, id=0161 l=5, 3225600010 tid=00
RX_MSG c=2, t=28393472, id=01F9 l=6, 201E18F4F8FF tid=00
RX_MSG c=2, t=28622848, id=0181 l=8, 18F4331032213F4E tid=00
RX_MSG c=2, t=30498816, id=0284 l=8, 0000000000000000 tid=00
RX_MSG c=2, t=30752768, id=0285 l=8, 0000000000000000 tid=00
RX_MSG c=2, t=32309248, id=01A5 l=8, DFFF012000000080 tid=00
RX_MSG c=2, t=38305792, id=0161 l=5, 3225600010 tid=00
RX_MSG c=2, t=38535168, id=0551 l=8, 776D6488FF7A0070 tid=00
RX_MSG c=2, t=38731776, id=01F9 l=6, 201E18F4F8FF tid=00
RX_MSG c=2, t=38961152, id=0181 l=8, 18F4331032213F4E tid=00
RX_MSG c=2, t=39198720, id=0511 l=7, 00000000000000 tid=00
```

Fig. 2: Recorded CAN messages

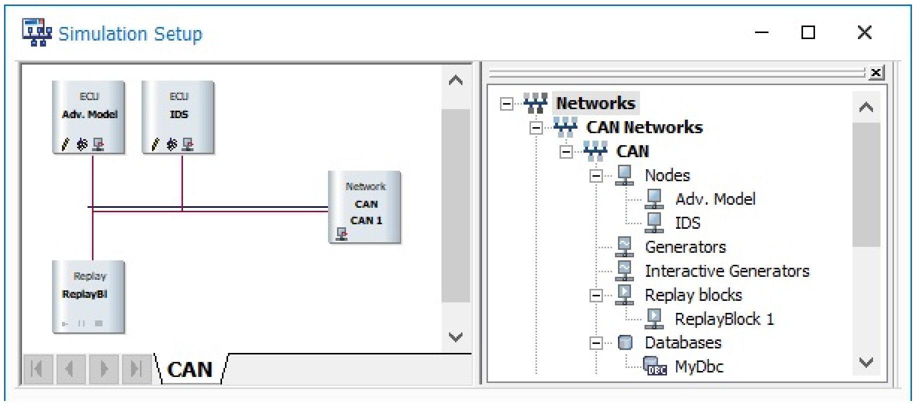


Fig. 3: CAN network architecture

CANoe provides us with all the building blocks for simulating and detecting real-world attacks on the CAN bus. The designed CAN analysis network from CANoe is depicted in Figure 3. This structure includes three blocks: a *replay node*, an *adversary model node* and the *IDS node*. The real traffic recorded from the vehicles is replayed in the simulation through a specialized type of node called a Replay block. For on-line attacks and analysis of vehicle traffic, the replay block can be simply disabled while connecting the CANoe simulated bus to the in-vehicle bus through a VN interfacing device. The adversary model node is implemented as a CAPL (CAN Application Programming Language) network node that mimics the behaviour of a real-world adversary. CAPL is a C-based language and provides additional CANoe specific functionalities, e.g., events, system variables, message structures and message databases. Finally, the overall traffic will be evaluated by an IDS node that is also programmed through CAPL.

## 3 Adversary Models

In this section we discuss the adversarial model that we account for and give a brief overview of its integration in CANoe.

### 3.1 Types of Attacks

In general, adversary models are based on the Dolev-Yao adversary which has full control over the network [5]. That is, the adversary can record, block, replay, modify or inject messages in the network. If any security mechanisms are in place, they are considered to be secure and the adversary can manipulate them only if he has the corresponding keys. In our work, we do not address security mechanisms since these are generally absent on the CAN traffic that we recorded and even if they are present we would not have access to manufacturer specifications (e.g., in case of authentication protocols over the CAN bus) since these are in general considered confidential information.

Our adversary has access to the entire traffic that was logged inside the vehicle. Based on existing literature on adversarial models for the CAN bus, our work considers the following types of attacks which we also integrate in the CANoe application:

1. **Replay of regular CAN frames** is the attack in which the adversary intercepts genuine frames and then replays them on the CAN bus. In this case the malicious frames are identical to genuine frames having the same identifier and data field. The only indicator for this type of attack is the frequency of the CAN messages (i.e., more frames with the same ID will be visible on the bus). The identifier of the attacked frame and the delay at which the attack frame is sent can be configured from the interface. The replay attacks can increase the busload which delays other frames or even aborts their transmission.
2. **Injection attacks** which consist in the insertion of adversarial frames on the bus and which we refine across the following lines:
   – **Injection of random data**, also referred in other works as fuzzy attacks [11], is an attack in which the adversary intercepts genuine frames and then injects the malicious frames on the CAN bus at a chosen delay after interception. The malicious frames have the same identifier as a genuine frame, but the data field is randomly generated. The delay is the time measured from the interception of the genuine frame event to triggering injection event of the attack frame. As in the previously described attack, our graphical user interface (GUI) allows for selecting the identifier of the targeted message. The transmission delay can be configured within 1 μs increments.
   – **Injection with scalar addition/multiplication of the datafield** - the data retrieved from in-vehicle sensors, e.g., speed sensor, engine temperature, steering angle, fuel pressure, brake pressure, are transmitted by network nodes via the CAN bus. Since sensors may have a linear transfer function, the slope of the function is a constant. This leads to attacks in which bytes of the CAN frames are incremented or multiplied by some constant values. Delays to the injected frames can be added as well.

– **Arbitrary injections** is the case in which the adversary can inject frames at will with the specified data or randomly generated data field and ID. In contrast to the previously defined attacks, the transmission of the injected frame will be done cyclically according to the configured cycle time.

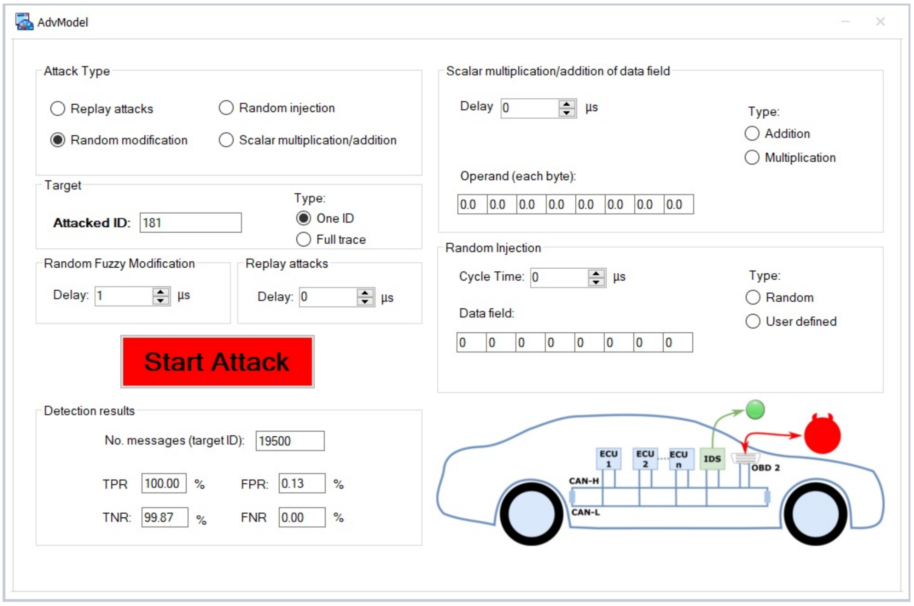
Other attacks have been also considered in the literature but are not included in our interface. In what follows, we explain why, at least for the moment, we did not considered them.

**DoS attacks** are trivial to mount on the CAN bus. Since the CAN ID is used in the arbitration mechanism to provide collision avoidance, continuously injecting messages with the highest priority ID, i.e., 0x000, leads to unavailability of the bus and the genuine frames are unable to transmit due to the loaded bus. However, detecting such an attack in which the ID 0x000 is sent in order to lock the bus would be trivial. For this, one can simply look for the consecutive occurrence of messages with this ID which does not show up in regular traces. A more sophisticated variant would be to send a low priority ID which is not null, but still has higher priority than regular IDs. This again can be detected trivially since the values of the genuine IDs are known by the manufacturer. Such attacks are accessible from the interface that we designed as *arbitrary injections* which allows to edit both the ID and data field but we do not view them separately as DoS attacks (which may be a consequence). The attack can be detected by the IDS, but the problem still remains since such attacks cannot be circumvented as high-priority IDs will win the bus anyway.

**Bus off attacks** are the adversarial action after which genuine nodes are placed in bus-off state. This can be done due to the error management system of CAN and such attacks are proved to be feasible by the works in [4] and [14]. Modeling such attacks may be of interest but our network is simulated based on existing traces and we don't have the specific behavior of the ECU implemented in the model. Moreover, such attacks can be circumvented only by modifying the error-handling mechanisms of CAN which is out of scope for the current work.

### 3.2 Application Interface

The application interface implemented in CANoe for allowing the configuration of the adversary node and IDS node is shown in Figure 4. We employ common controls, e.g., radio buttons, combo box, to provide an user friendly interface. The relationship between the graphical interface and CAPL is made through system variables since they can be retrieved by specific CAPL functions and events. Consequently, our adversary model has the benefit of providing various types of attacks and can perform the following actions: read, modify and replay messages. In the first step, the user must select the type of attack that will be used. Another option allows the user to select if the attack should target a single specified ID or all messages in the trace. For each type of attack, specific parameters can be configured. On the other hand, during the simulation run, the detection algorithm is running on the IDS node to classify frames. The indicator led will switch to either green or red depending on genuine or malicious received frame. Moreover, at the end of the simulation the results of the detection rates and the number of the targeted messages are presented.


AdvModel

Attack Type

Replay attacks Random injection [✓ Random modification] Scalar multiplication/addition

Target

**Attacked ID:** 181

Type: [✓ One ID] Full trace

Random Fuzzy Modification

Delay: 1 µs

Replay attacks

Delay: 0 µs

Start Attack

Detection results

No. messages (target ID): 19500

TPR 100.00 % FPR: 0.13 %

TNR: 99.87 % FNR 0.00 %

Scalar multiplication/addition of data field

Delay 0 µs

Type: Addition Multiplication

Operand (each byte): 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0

Random Injection

Cycle Time: 0 µs

Type: Random User defined

Data field: 0 0 0 0 0 0 0 0

Fig. 4: Graphical interface for the designed application

## 4 Intrusion Detection Algorithms

In this section we discuss about the tools used in our evaluation and the Matlab-CANoe integration. We also discuss some background on the k-NN algorithm which we use for intrusion detection.

### 4.1 Statistics and Machine Learning Toolbox

For implementing the intrusion detection mechanism we employ Matlab, namely, the *statistics and machine learning toolbox* made available by the framework. This toolbox provides a range of machine learning algorithms for solving regression or classification problems. These algorithms are based on either supervised or unsupervised learning and we choose k-NN since it is a commonly employed solution when little is known about the input data. Indeed, in our case the data comes from traces that were logged inside vehicles and we don't have any access to the manufacturer's requirements. Consequently, there is no prior knowledge on the data, but we can label the malicious CAN frames that we inject for the training trace. In the training phase, the supervised learning (employed also by the k-NN algorithm) has as observation samples, a collection of n pairs $\{(i_0, o_0), (i_1, o_1), \ldots (i_{n-1}, o_{n-1})\}$, which consists of the inputs and the desired outputs. The output of the training phase is a model (a trained function) responsible for predictions over new data that will be given in the test phase.

We also took advantage of Matlab's capability to generate C/C++ code with the trained model and prediction function. We build a dynamic library (dll) based on this

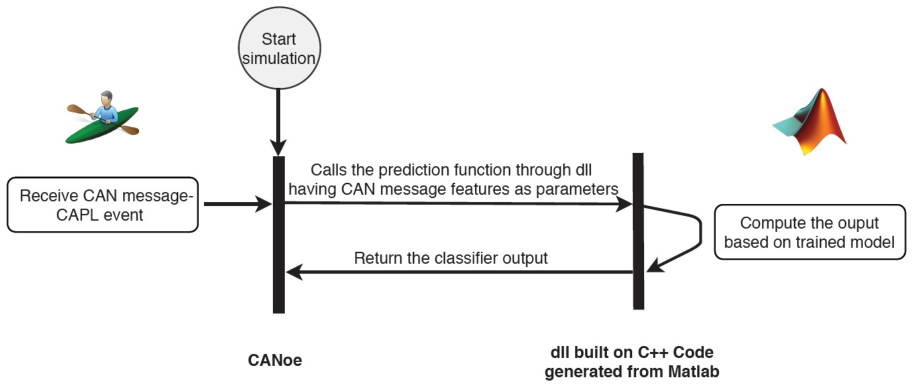

Start simulation
Receive CAN message-CAPL event
Calls the prediction function through dll having CAN message features as parameters
Compute the ouput based on trained model
Return the classifier output
CANoe
dll built on C++ Code generated from Matlab

Fig. 5: The flowcart of the data exchanged between Matlab and CANoe through dll

code and integrate the functionality in CANoe through CAPL code. The integration of a custom library into CANoe provides the advantage of accessing system resources, e.g, CPU, memory [22], which are otherwise not direclty available in CANoe. Figure 5 illustrates the interaction between CANoe and the Matlab-based library for analyzing CAN messages.

## 4.2 k-NN Algorithm

We use the k-NN algorithm as a basis for our evaluation. This algorithm is commonly employed in classification problems and even in network IDS [20]. The k-NN uses a distance metric, e.g., the Euclidean, Hamming, Minkowski, Jaccard distances, etc. For most of our analysis we choose the Euclidean distance but it is easy to switch to any of the previously mentioned.

In general, a machine-learning algorithm has two stages: the training stage and the testing stage. Consequently, we split the CAN trace into a training and a testing part. In our experiments, the first stage is performed offline with the purpose of training the classifier based on inputs-output pairs. In this stage, each input is mapped to the true class $c$ (genuine or malicious frame). The end of this stage outputs the k-NN model. The second stage is the real-time detection based on the trained model. In this stage, each input is mapped to the predicted class $\hat{c}$ based on the decision rule. The decision rule depends on the number of neighbors $k$ as follows:

1. Decision rule when *k=1*: let $m_t$ be a test frame and $m_i$ a training frame, then $m_n$ is nearest neighbor to $m_t$ if and only if the Euclidean distance: $d_e(m_t, m_n) = min_i\{d_e(m_t, m_i)\}$, where $i$ covers the range of training frames. The predicted response of $\hat{c}$ from the trained model will be equal with the true class $c$ of the $m_i$ which has the minimum Euclidean distance to $m_t$.
2. Decision rule when *k>1*: The predicted response $\hat{c}$ of the $m_t$ from the trained model will be equal with the most encountered $c$, through the k nearest training messages.

The k-NN input observation is a vector that accounts for the data field and the delay between consecutive timestamps of the same ID. In such case, the input sample $I \in \{0,1\}^9$ is described mathematically as follows: $I = \big\{i_0, i_1, i_2, ..., i_8\big\}$, where $i_0$ represents the delay and $i_1...i_8$ represent each byte from the data field. We choose an odd number of neighbors (e.g. 1, 3, 15) in order to avoid an equal number of votes and select a majority.

## 5 Experimental Results

We first discuss the metrics employed for evaluating the intrusion detection algorithms, then we proceed to presenting the experimental results.

### 5.1 Metrics for Evaluating the Performance of the IDS

Since our evaluation performs a binary classification of the CAN frames, we measure the performance of the IDS based on the most commonly four metrics:

1. the sensitivity or the true positive rate - measures the percentage of the CAN frames that are correctly classified as malicious, i.e., $TPR = TP/(TP + FN)$.
2. the false negative rate- measures the percentage of the CAN frames that are reported as genuine frames but are actually malicious frames, i.e., $FNR = FN/(FN+TP)$.
3. the specificity or the true negative rate - measures the percentage of the CAN frames that are correctly classified as genuine, i.e., $TNR = TN/(TN + FP)$.
4. fall-out or the false positive rate - measures the percentage of the CAN frames that are reported as malicious, but the true class of the frames is genuine, i.e., $FPR = FP/(FP + TN)$.

### 5.2 Results on Detection Accuracy

We devise our experiments to cover the previously defined adversarial models. For each type of attack, we have different scenarios depending on the delay of the attack frame. Multiplication or addition coefficients may be also applied to the data field. Since the traces we obtained from vehicles did not contain extended frames, we experiment only with standard frames. We build our datasets using the CANoe simulation by injecting malicious frames on a single targeted CAN ID or over the full trace, i.e., all CAN IDs. The results obtained for detecting attacks on a single CAN ID are based on portions of traces containing 500 frames used for training and 19500 frames for the actual tests. We choose only a small percent for training to cover the more realistic scenario where the IDS is trained for a limited time, e.g., during production, and then runs for a longer period. In the current experiments (on a single CAN ID) we have only attacked frames that have a cycle time of 10 ms since this is a very common periodicity, but similar results will be likely obtained for other delays. For the full trace attacks we employ 5000 training frames and 45000 test frames.

We now discuss the results on replay attacks which are presented in Table 1. Extended results for this scenario are deferred to Table 2 from Appendix 2. In this case the

Table 1: Detection rates for various types of attacks

| | Attack params. | | | No. messages | | k-NN Parameters | | Detection rates | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| No. | Att. type | Operand | Delay ms | training | testing | No. neigh. | Distance | TNR | TPR | FPR | FNR |
| 1. | r | $n/a$ | 9.750 | 500 | 19500 | 1 | Euclidean | 99.00% | 99.65% | 1.00% | 0.35% |
| 2. | r | $n/a$ | 0.001 | 500 | 19500 | 1 | Euclidean | 100% | 100% | 0% | 0% |
| 3. | r | $n/a$ | 0.001 | 500 | 19500 | 1 | Euclidean | 88.86% | 100% | 11.14% | 0% |
| 4. | r | $n/a$ | 5 | 500 | 19500 | 1 | Euclidean | 88.88% | 100% | 11.12% | 0% |
| 5. | r | $n/a$ | 9 | 500 | 19500 | 1 | Euclidean | 90.33% | 83.47% | 9.67% | 16.53% |
| 6. | r | $n/a$ | 9.750 | 500 | 19500 | 1 | Euclidean | 87.98% | 51.88% | 12.02% | 48.12% |
| 7. | r | $n/a$ | 50 | 500 | 19500 | 1 | Euclidean | 88.31% | 84.66% | 11.69% | 15.34% |
| 8. | ir | $n/a$ | 0.001 | 500 | 19500 | 1 | Euclidean | 99.87% | 100% | 0.13% | 0% |
| 9. | ir | $n/a$ | 9.750 | 500 | 19500 | 1 | Euclidean | 99.87% | 100% | 0.13% | 0% |
| 10. | isa | $\alpha = 2$ | 0.001 | 500 | 19500 | 1 | Euclidean | 89.63% | 100% | 10.37% | 0% |
| 11. | isa | $\alpha = 2$ | 9.750 | 500 | 19500 | 1 | Euclidean | 91.34% | 53.98% | 8.66% | 46.02% |
| 12. | ism | $\alpha = 2$ | 0.001 | 500 | 19500 | 1 | Euclidean | 89.65% | 100% | 10.35% | 0% |
| 13. | ism | $\alpha = 2$ | 9.750 | 500 | 19500 | 1 | Euclidean | 91.38% | 67.74% | 8.62% | 32.26% |
| 14. | isa | $\alpha = 2$ | 9.750 | 500 | 19500 | 1 | E($\Delta t$), H(data) | 90.72% | 100% | 9.28% | 0% |
| 15. | ism | $\alpha = 2$ | 9.750 | 500 | 19500 | 1 | E($\Delta t$), H(data) | 90.75% | 85.87% | 9.25% | 14.13% |
| 16. | r | $n/a$ | 0.001 | 5000 | 45000 | 1 | Euclidean | 95.21% | 100% | 4.79% | 0% |
| 17. | r | $n/a$ | 5 | 5000 | 45000 | 1 | Euclidean | 95.45% | 100% | 4.55% | 0% |
| 18. | r | $n/a$ | 9.750 | 5000 | 45000 | 1 | Euclidean | 95.23% | 66.58% | 4.77% | 33.42% |
| 19. | r | $n/a$ | $\{$9.75, 19.75, 39.75, 99.75$\}$ | 5000 | 45000 | 1 | Euclidean | 94.76% | 50.06% | 5.24% | 49.94% |
| 20. | ir | $n/a$ | 0.001 | 5000 | 45000 | 1 | Euclidean | 99.53% | 100% | 0.47% | 0% |
| 21. | ir | $n/a$ | 5 | 5000 | 45000 | 1 | Euclidean | 99.40% | 100% | 0.6% | 0% |
| 22. | ir | $n/a$ | 9.750 | 5000 | 45000 | 1 | Euclidean | 99.57% | 91.52% | 0.43% | 8.48% |

training phase was performed on traces that contain regular frames and replay attack frames sent at a 9.750 ms (row 1 from Table 1, rows 1-2 from Table 2) and 0.001 ms (row 2 from Table 1, rows 3-4 from Table 2) delay after the genuine frame. The first delay is chosen specifically for the attack frame to arrive just before the genuine frame on the bus (the genuine frame will arrive periodically at 10 ms and $\approx 250\mu s$ is the physical time of the frame on the bus) while the second is to assure that the attack frame arrives immediately after the genuine frame. We use both the content of the datafield and the delay between consecutive timestamps of the targeted CAN ID ($\Delta t$) as inputs for the training phase. The detection rates were 100% in case of 0.001 ms delay and around 99% in case of 9.750 ms while the false positive rate is 0% in the first scenario and around 1% in the second. There is a slight increase of false positive rate in the first scenario since in case of the 9.750 ms delay, the injected frames are sent very close to the transmission time of genuine frames. Consequently, in some cases the legit frame is mismatched for the attack frame. The good detection result is also due to the less realistic assumption that an attacker will send all its frames with the fixed delay that was used in the training phase.

Thus the next step in our evaluation, was to train the classifier based on one delay, i.e., 9.750 ms while the evaluation frames were built with other delay, i.e., 9 ms. As expected, the detection rate drops under 20%. Consequently, to overcome this problem, we chose to train the classifier based on traces built with replay injections at a random delay covering the whole range between 0 and the cycle time of the frame, since the IDS must be able to detect attacks frames sent with any delay. All the results that follow are based on such randomized delays. We present the results for this scenario in Table 1 (rows 3-7) and their extension is deferred to Table 2 (rows 5-14) from Appendix 2. In case of 0.001 ms and 5 ms delays, the true positive rate is close to 100% while the false positive rate is around 10%. The false positive rates are caused by the identical data field of regular and injected frames.

In the next two attack scenarios (rows 5-6 from Table 1, rows 9-12 from Table 2) the adversarial actions are more refined and well thought out. These actions are designed so that the injected message is sent on the bus shortly before, i.e., 9ms delay, or even close enough to overlap with the genuine message in some cases, i.e., 9.750 ms delay. The detection rate degrades to the point that the TPR drops to below 80% for the first case and around 50% for the second. What can also be observed, from the majority of the results, is that with the growth in the number of neighbors comes a slight increase in specificity and a decrease in sensitivity, which is sometimes more pronounced, i.e., from 83% (row 5 from Table 1) to 52% (row 9 from Table 2).

The results obtained on injections with random data are shown in Table 1 (rows 8-9) while the extension of the results is presented in Table 2 (rows 15-18) from Appendix 2. In this case we obtained detection rates close to 100% percents for both of the tested delay scenarios. We also observe a negligible amount of false positives. The high detection rate is justified by the high entropy of the injected frames data field that differs from the authentic messages. In general, this type of attacks is much easier to detect than replay attacks.

As expected, results for injection attacks using scalar addition or multiplication, presented in Table 1 (rows 10-15) and the extension in Table 2 from Appendix 2 (rows 19-30), exhibit a lower detection rate especially as we used a very low value for the scalar (thus modifications of the datafield are small). At a first view, the results are very similar to those obtained for replay attacks for the same delays: 0.001 ms (row 3 from Table 1 and rows 5-6 from Table 2) and 9.750 ms (row 6 from Table 1 and rows 11-12 from Table 2). This can be explained by the message periodicity having a greater influence on the result of the prediction function than the data field. This happens since the operation of adding $\alpha = 2$ to each byte of the data field does not have a considerable effect on the Euclidean distance. We chose $\alpha = 2$ to assure only a small change in the message (obviously, a larger $\alpha$ will lead to more modifications and will be easier to detect). In case of scalar multiplication the detection rate increases to around 67% (row 13 from the Table 1) since the operation of scalar multiplication with $\alpha = 2$ has a greater impact on the resulting Euclidean distance.

A better approach to improve the detection results, is the use of two trained models: the first trained with $\Delta t$ based on the Euclidean distance and the second trained based on the data field using the Hamming distance. In this case, each model classifier predicts a class for each message. Denoting the predicted class for the first model as $\hat{c}_1$ and the second one as $\hat{c}_2$, the final predicted class $\hat{c}$ is : $\hat{c} \in \hat{c}_1 \vee \hat{c}_2$. This approach improves the sensitivity to 100% in case of scalar addition and at 85% in case of multiplication while the false positive rate remains around the 10% level as can be seen in Table 1 (rows 14-15). By E($\Delta t$) and H(data) we denote the Euclidean and Hamming distances on the delay and data respectively.

The next step in improving detection capabilities consists in covering the full trace since monitoring a single ID would involve one trained model for each CAN ID and leads to the need for large computational/memory resources which may not be available.

The full trace contains frames having 10 ms, 20 ms, 40 ms, and 100 ms cycle times. The full attack trace was build as following. We define the attack probability for each frame as a constant $P_r(A)$. A variable $\epsilon \in [0, 100]$ is randomly generated and if $\epsilon$ is less than or equal to $P_r(A)$, then the frame is attacked otherwise it is left unaltered. For our experiments we configured $P_r(A) = 30$. Therefore, the input in our classifier accounts for the CAN ID before the $\Delta t$ and data field.

The results over the full trace for replay attacks, are presented in Table 1 (rows 16-19), Table 2 from Appendix 2 (rows 31-38), and rows 20-22 from Table 1, rows 39-44 from Table 2, for fuzzy attacks. Even if for an extended evaluation, with a single trained model, the results remain satisfactory. In case of replay attacks, the detection rates are similar (for 0.001 ms and 5 ms delays) or even better (for 9.750 ms delays) than those obtained for a single ID. This happens since the attack frame that has 9.750 ms delay is sent on the bus ahead or even overlaps with the authentic frame just in case of 10 ms cycle, while the full trace contains more cycle times values for which the attack frame is even more conspicuous. A cleverer adversary may of course choose delays that are closer to the cycle time of each frame. This scenario is presented in the row 18 of Table 1 and rows 37-38 of Table 2. The detection rate is approximately two percents lower than monitoring for a single ID in case of the directed replay attacks (around 50%). For random attacks, the sensitivity is most of the part close to 100%, except for the 9.750 ms delay, where it drops to around to 90% in case of using one neighbor and to 60% when more neighbors are employed.

## 6 Conclusion

Our work explores the integration of adversary models and intrusion detection systems in CANoe. Since adversarial actions are modeled over real-world in-vehicle traces, the results offer a more realistic testbed for in-vehicle network attacks. As future work it would be of interest to allocate specific parts of the traffic to a particular ECU which would allow targeted attacks toward specific ECUs. A complete simulation for the behavior of each ECU is a more complex goal but perhaps achievable in the future. Adversarial actions are easier to test inside a simulation environment and the risk for damaging the actual car is removed. Adding MATLAB functionalities for machine-learning in order to classify CAN packets is a convenient way for designing and testing such an IDS due to the rich machine learning toolset offered by MATLAB. Adding other algorithms for intrusion detection is an immediate goal for extending our framework.

**Acknowledgement.** This work was supported by a grant of the Romanian Ministry of Research and Innovation, CNCS - UEFISCDI, project number PN-III-P1-1.1-PD-2016-1198, within PNCDI III.

## References

1. O. Y. Al-Jarrah, C. Maple, M. Dianati, D. Oxtoby, and A. Mouzakitis. Intrusion Detection Systems for Intra-Vehicle Networks: A Review. *IEEE Access*, 7:21266–21289, 2019.

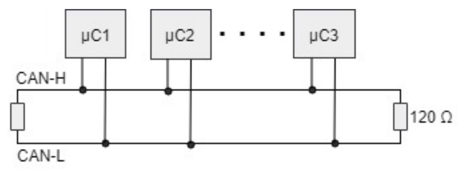
2. A. Alshammari, M. A. Zohdy, D. Debnath, and G. Corser. Classification Approach for Intrusion Detection in Vehicle Systems. 2018.
3. S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, volume 4, pages 447–462. San Francisco, 2011.
4. K.-T. Cho and K. G. Shin. Error handling of in-vehicle networks makes them vulnerable. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1044–1055. ACM, 2016.
5. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.
6. C. E. Everett and D. McCoy. {OCTANE}(Open Car Testbed and Network Experiments): Bringing Cyber-Physical Security Research to Researchers and Students. In *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test*, 2013.
7. B. Groza and P.-S. Murvay. Efficient Intrusion Detection With Bloom Filtering in Controller Area Networks. *IEEE Transactions on Information Forensics and Security*, 14(4):1037–1051, 2019.
8. T. Hoppe, S. Kiltz, and J. Dittmann. Security threats to automotive CAN networks—Practical examples and selected short-term countermeasures. *Reliability Engineering & System Safety*, 96(1):11–25, 2011.
9. M.-J. Kang and J.-W. Kang. Intrusion detection system using deep neural network for in-vehicle network security. *PloS one*, 11(6):e0155781, 2016.
10. P. Kleberger, T. Olovsson, and E. Jonsson. Security aspects of the in-vehicle network in the connected car. In *2011 IEEE Intelligent Vehicles Symposium (IV)*, pages 528–533. IEEE, 2011.
11. H. Lee, S. H. Jeong, and H. K. Kim. Otids: A novel intrusion detection system for in-vehicle network by using remote frame. *Privacy, Security and Trust (PST) 2017*, 2017.
12. M. Marchetti, D. Stabili, A. Guido, and M. Colajanni. Evaluation of anomaly detection for in-vehicle networks through information-theoretic algorithms. In *Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, pages 1–6. IEEE, 2016.
13. C. Miller and C. Valasek. Adventures in automotive networks and control units. *Def Con*, 21:260–264, 2013.
14. P.-S. Murvay and B. Groza. DoS Attacks on Controller Area Networks by Fault Injections from the Software Layer. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, pages 71:1–71:10, 2017.
15. M. Müter and N. Asaj. Entropy-based anomaly detection for in-vehicle networks. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pages 1110–1115. IEEE, 2011.
16. S. N. Narayanan, S. Mittal, and A. Joshi. OBD_SecureAlert: An Anomaly Detection System for Vehicles. In *Smart Computing (SMARTCOMP), 2016 IEEE International Conference on*, pages 1–6. IEEE, 2016.
17. D. K. Nilsson, U. E. Larson, F. Picasso, and E. Jonsson. A first simulation of attacks in the automotive network communications protocol flexray. In *Proceedings of the International Workshop on Computational Intelligence in Security for Information Systems CISIS'08*, pages 84–91. Springer, 2009.
18. J. Petit and S. E. Shladover. Potential cyberattacks on automated vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):546–556, 2014.
19. D. Stabili, M. Marchetti, and M. Colajanni. Detecting attacks to internal vehicle networks through Hamming distance. In *2017 AEIT International Annual Conference*, pages 1–6. IEEE, 2017.
20. M.-Y. Su. Real-time anomaly detection systems for denial-of-service attacks by weighted k-nearest-neighbor classifiers. *Expert Systems with Applications*, 38(4):3492–3498, 2011.

21. D. Tian, Y. Li, Y. Wang, X. Duan, C. Wang, W. Wang, R. Hui, and P. Guo. An intrusion detection system based on machine learning for CAN-bus. In *International Conference on Industrial Networks and Intelligent Systems*, pages 285–294. Springer, 2017.
22. Vector. *CAPL DLL Description*, 2007.

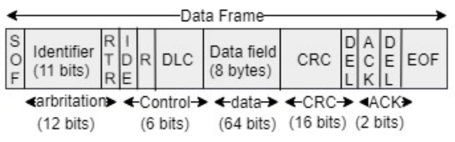
## Appendix 1 - Brief Description of the CAN bus

CAN provides bit rates of up to 125 kbit/s for low-speed CAN and up to 1 Mbit/s on high-speed CAN while carrying up to 8 bytes of payload. Increased communication speeds and payloads of up to 64 bytes are possible by using the CAN-FD (CAN- Flexible Data) protocol extension.

At the physical layer, CAN is implemented as a two wire (CAN-high and CAN-low) differential bus. A common CAN network topology is shown in Figure 6 a). The main communication element used by CAN is the data frame with a structure as presented in Figure 6 b). The data frame is received by all ECUs but it is only used by ECUs interested in its content for processing purposes. This frame filtering is usually done based on the CAN ID (identifier). The ID also serves for assuring packet arbitration as part of the collision avoidance mechanism which gives higher priority to frames with lower ID values in the case two frames are simultaneously transmitted. The CAN ID can be either 11 bits long (in standard frames), or 29 bits (in extended format).



(a)



(b)

Fig. 6: CAN network topology (a) and data frame format (b)

## Appendix 2 - Results for Various Number of Neighbors over a Single ID and over Full Trace

Table 2: Detection rates for various types of attacks (k-NN with 3 or 15 neighbors)

| | Attack params. | | | No. messages | | k-NN Parameters | | Detection rates | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| No. | Att. type | Operand | Delay ms | training | testing | No. neigh. | Distance | TNR | TPR | FPR | FNR |
| 1. | r | $n/a$ | 9.750 | 500 | 19500 | 3 | Euclidean | 98.55% | 99.31% | 1.45% | 0.69% |
| 2. | r | $n/a$ | 9.750 | 500 | 19500 | 15 | Euclidean | 97.55% | 97.83% | 2.45% | 2.17% |
| 3. | r | $n/a$ | 0.001 | 500 | 19500 | 3 | Euclidean | 100% | 100% | 0% | 0% |
| 4. | r | $n/a$ | 0.001 | 500 | 19500 | 15 | Euclidean | 100% | 100% | 0% | 0% |
| 5. | r | $n/a$ | 0.001 | 500 | 19500 | 3 | Euclidean | 90.02% | 100% | 9.98% | 0% |
| 6. | r | $n/a$ | 0.001 | 500 | 19500 | 15 | Euclidean | 91.32% | 100% | 8.68% | 0% |
| 7. | r | $n/a$ | 5 | 500 | 19500 | 3 | Euclidean | 89.99% | 100% | 10.01% | 0% |
| 8. | r | $n/a$ | 5 | 500 | 19500 | 15 | Euclidean | 91.30% | 100% | 8.70% | 0% |
| 9. | r | $n/a$ | 9 | 500 | 19500 | 3 | Euclidean | 91.61% | 52.53% | 8.39% | 47.47% |
| 10. | r | $n/a$ | 9 | 500 | 19500 | 15 | Euclidean | 91.33% | 31.11% | 8.67% | 68.89% |
| 11. | r | $n/a$ | 9.750 | 500 | 19500 | 3 | Euclidean | 89.33% | 50.67% | 10.67% | 49.33% |
| 12. | r | $n/a$ | 9.750 | 500 | 19500 | 15 | Euclidean | 91.26% | 50.67% | 8.74% | 49.33% |
| 13. | r | $n/a$ | 50 | 500 | 19500 | 3 | Euclidean | 89.96% | 83.75% | 10.04% | 16.25% |
| 14. | r | $n/a$ | 50 | 500 | 19500 | 15 | Euclidean | 91.40% | 83.75% | 8.60% | 16.25% |
| 15. | ir | $n/a$ | 0.001 | 500 | 19500 | 3 | Euclidean | 99.70% | 100% | 0.30% | 0% |
| 16. | ir | $n/a$ | 0.001 | 500 | 19500 | 15 | Euclidean | 98.63% | 100% | 1.37% | 0% |
| 17. | ir | $n/a$ | 9.750 | 500 | 19500 | 3 | Euclidean | 99.70% | 100% | 0.30% | 0% |
| 18. | ir | $n/a$ | 9.750 | 500 | 19500 | 15 | Euclidean | 98.63% | 100% | 1.37% | 0% |
| 19. | isa | $\alpha = 2$ | 0.001 | 500 | 19500 | 3 | Euclidean | 91.37% | 100% | 8.63% | 0% |
| 20. | isa | $\alpha = 2$ | 0.001 | 500 | 19500 | 15 | Euclidean | 91.13% | 100% | 8.87% | 0% |
| 21. | isa | $\alpha = 2$ | 9.750 | 500 | 19500 | 3 | Euclidean | 92.15% | 51.01% | 7.85% | 48.99% |
| 22. | isa | $\alpha = 2$ | 9.750 | 500 | 19500 | 15 | Euclidean | 91.09% | 50.40% | 8.91% | 46.60% |
| 23. | ism | $\alpha = 2$ | 0.001 | 500 | 19500 | 3 | Euclidean | 91.39% | 100% | 8.61% | 0% |
| 24. | ism | $\alpha = 2$ | 0.001 | 500 | 19500 | 15 | Euclidean | 91.12% | 100% | 8.88% | 0% |
| 25. | ism | $\alpha = 2$ | 9.750 | 500 | 19500 | 3 | Euclidean | 92.16% | 60.70% | 7.84% | 39.30% |
| 26. | ism | $\alpha = 2$ | 9.750 | 500 | 19500 | 15 | Euclidean | 91.11% | 50.59% | 8.89% | 49.41% |
| 27. | isa | $\alpha = 2$ | 9.750 | 500 | 19500 | 3 | E($\Delta t$), H(data) | 91.07% | 100% | 8.93% | 0% |
| 28. | isa | $\alpha = 2$ | 9.750 | 500 | 19500 | 15 | E($\Delta t$), H(data) | 91.06% | 100% | 8.94% | 0% |
| 29. | ism | $\alpha = 2$ | 9.750 | 500 | 19500 | 3 | E($\Delta t$), H(data) | 91.06% | 85.52% | 8.94% | 14.48% |
| 30. | ism | $\alpha = 2$ | 9.750 | 500 | 19500 | 15 | E($\Delta t$), H(data) | 91.09% | 85.17% | 8.91% | 14.83% |
| 31. | r | $n/a$ | 0.001 | 5000 | 45000 | 3 | Euclidean | 96.74% | 100% | 3.26% | 0% |
| 32. | r | $n/a$ | 0.001 | 5000 | 45000 | 15 | Euclidean | 98.77% | 100% | 1.23% | 0% |
| 33. | r | $n/a$ | 5 | 5000 | 45000 | 3 | Euclidean | 96.75% | 100% | 3.25% | 0% |
| 34. | r | $n/a$ | 5 | 5000 | 45000 | 15 | Euclidean | 98.70% | 100% | 1.30% | 0% |
| 35. | r | $n/a$ | 9.750 | 5000 | 45000 | 3 | Euclidean | 96.79% | 65.33% | 3.21% | 34.67% |
| 36. | r | $n/a$ | 9.750 | 5000 | 45000 | 15 | Euclidean | 98.83% | 35.95% | 1.17% | 64.05% |
| 37. | r | $n/a$ | $\{$9.75, 19.75, 39.75, 99.75$\}$ | 5000 | 45000 | 3 | Euclidean | 96.40% | 48.94% | 3.60% | 51.06% |
| 38. | r | $n/a$ | $\{$9.75, 19.75, 39.75, 99.75$\}$ | 5000 | 45000 | 15 | Euclidean | 98.68% | 47.41% | 1.32% | 52.59% |
| 39. | ir | $n/a$ | 0.001 | 5000 | 45000 | 3 | Euclidean | 99.47% | 100% | 0.53% | 0% |
| 40. | ir | $n/a$ | 0.001 | 5000 | 45000 | 15 | Euclidean | 99.57% | 100% | 0.43% | 0% |
| 41. | ir | $n/a$ | 5 | 5000 | 45000 | 3 | Euclidean | 99.31% | 100% | 0.69% | 0% |
| 42. | ir | $n/a$ | 5 | 5000 | 45000 | 15 | Euclidean | 99.49% | 100% | 0.51% | 0% |
| 43. | ir | $n/a$ | 9.750 | 5000 | 45000 | 3 | Euclidean | 99.53% | 86.46% | 0.47% | 13.54% |
| 44. | ir | $n/a$ | 9.750 | 5000 | 45000 | 15 | Euclidean | 99.63% | 60.97% | 0.37% | 39.03% |