UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO MATEMATICA

UNIVERSITÀ DEGLI STUDI DI PADOVA

MASTER DEGREE IN CYBERSECURITY

COURSE: CYBER-PHYSICAL SYSTEMS AND IOT SECURITY

Report for the Course on Cyber-Physical Systems and IoT Security

Reference Paper: **Error Handling of In-vehicle Networks Makes Them Vulnerable**

Report and Project:

Jacopo Momesso 2123874

Academic year 2024/2025

Contents

1 Objectives

The reference paper "Error Handling of In-Vehicle Networks Makes Them Vulnerable" discusses the discovery of a new vulnerability in in-vehicle networks that makes them vulnerable to a new type of attack: the bus-off attack. Therefore, the main goal of the project is to simulate the bus-off attack. However, in order to simulate the attack, a simulator of the CAN bus and the ECU that make up the in-vehicle network is required; otherwise, the attack cannot be exploited. Thus, the project has two goals:

- Build a simulator capable of simulating a simplified CAN bus and the ECUs connected to it that can send messages over the bus
- Simulate the bus-off attack with the simulator

2 System Setup

The tool I used in this project is Python (version 3.12.8, but python 3.11 should also be acceptable), and three of its libraries:

- python.time
- python.threading
- python.matplotlib.pyplot

With Python installed, to run the simulator should be sufficient to open the terminal and type:

```
python3 simulator_name.py
```

or

```
python simulator_name.py
```

If it doesn't work, I created a virtual Python environment in my OS using the following terminal command:

```
python -m venv /path/to/new/virtual/environment
```

Once the virtual environment was created, I activated it using:

```
source ./path/to/new/virtual/environment/bin/activate
```

Finally, running the simulator should not give any errors.

3 Experiments

The simulator has 3 different classes that represent:

- the packets sent over the CAN bus
- the ECUs (Electronic Control Unit)
- the CAN bus (Control Area Network)

```
# Create the CAN bus and ECU objects
canbus = CANBus()
ecu1 = ECU(1)
ecu2 = ECU(2, attack_mode=ADVERSARY_ECU, adv_target=1)
# Connect the ECUs to the CAN bus
canbus.connect_ecu(ecu1)
canbus.connect_ecu(ecu2)
# Create a barrier for synchronizing two thread
barrier = threading.Barrier(2)
# Start two thread for the two ECUs
thread1 = threading.Thread(target=run_ecu, args=(ecu1, 100, [1, 0, 1], canbus, barrier))
thread2 = threading.Thread(target=run_ecu, args=(ecu2, 100, [0, 0, 1], canbus, barrier))

thread1.start()
thread2.start()

thread1.join()
thread2.join()
# Plot the final TEC history graph
plot_tec_history(ecu1, ecu2)
```

Figure 1: Creation of the simulator's part

Along with the 3 classes there're two functions, one that simulate the ECU behaviour, and one for plotting the graph of the final results of the bus-off attack.
The program creates the CAN bus, two ECUs (one victim and one adversary) connected to the CAN bus, for simulating the bus-off attack, and then two different threads for running the two ECUs simultaneously, making the whole program more realistic (Figure 1).

3.1 Packet class

```
# Class representing a simplified CAN packet
class Packet:
    def __init__(self, ecu_id, id, rtr, data, control=CONTROL, sof=SOF, ack=ACK, eof=EOF, crc=CRC):…

    def to_Array(self):…

    def get_EcuId(self):…

    def get_Id(self):…

    def get_Control(self):…

    def get_Data(self):…
```

Figure 2: Packet class and its methods

The Packet class (Figure 2) is a simplified representation of a packet that the ECUs send over the CAN bus. As a real packet, it has all its sections:

- SoF (Start Of Frame) and EoF (End of Frame); for simplicity the are both 1
- ID; for simplicity is an integer value
- RTR (Remote Transmission Request); for simplicity is 0 when the ECU requests data, and 1 when it sends data

- Control; for simplicity I decided to use the 1 value if the ECU is not the adversary one, while 0 if the ECU is the adversary
- Data; it's an array of integer
- CRC (Cyclic Redundancy Check); for simplicity I decided to put its value always to 0
- ACK (Acknowledge field); for simplicity I decided to put its value always to 0

Along with these parameters, the packet class also has a parameter that identifies the ECU (ecu_id) that sends the packet.
The main function of the class is *to_Array()*, which returns the packet as an array, which is easier to compare when the simulator has to check two of them. The other functions simply return some specific parameters of the class, such as the ID, data and so on.

3.2 ECU class

```
# Class representing a simplified ECU (Electronic Control Unit)
class ECU:
    def __init__(self, ecu_id, attack_mode=NORMAL_ECU, adv_target=-1):…

    def ecu_status_check(self):…

    def send_packet(self, packet_id, data, canbus, retransmission=False):…

    def check_duplicate_ids(self, packet_id, canbus):…

    def tec_rec_check(self, packet_id, canbus):…
```

Figure 3: ECU class and its methods

The ECU class (Figure 3) represents a simplified ECU. Like a real ECU, the class has its own TEC and REC to handle the presence of errors during packet transmission, and it also has its own status, which is always Error Active at the start, but can change to Error Passive or Bus-Off when TEC and REC change.
In fact, the *ecu_status_check()* function checks the value of TEC and REC and, depending on their values, changes the status of the ECU and the retransmission delay (a simplified way to simulate the difference in bit needed to retransmit the packet while the ECU is in Error Active and in Error Passive). Moreover, thanks to the *send_packet()* function, the ECU can send packets, check if at the same moment packets with the same ID are being sent on the CAN bus using the *check_duplicate_ids()* function, and finally increment or decrement TEC and REC according to the presence or not of errors or flags thanks to the *tec_rec_check()* function.
More specifically, the *check_duplicate_ids()* function looks for packets with the same ID in the packet_log array on the CAN bus. If it finds duplicate packets, it checks whether the two packets are different (the bus-off attack relies on a dominant bit mismatch in the adversary's packet). If the packets are different, i.e. there's a bit mismatch, it sets its own retrasmission_needed variable to True, since the packet needs to be retransmitted. Then the function checks the status of the ECU, and based on that it does other checks. If the ECU is in an error passive state and it's not the adversary's ECU, it sets the passive flag

to True (it represents the presence of the flag due to an error). If the ECU is an adversary one, it does nothing, in fact during a bus-off attack, the adversary ECU once it enters error passive state does not send any flag as the victim ECU sends a passive flag. If the ECU is in error active, it sets the active flag (it represents the presence of the flag due to an error) to True.
The *tec_rec_check()* function updates the TEC and REC values according to the status and to the prensence or not of the flags setted by the *check_duplicate_ids()* function. As the latter, it looks for duplicates and when present, it checks for difference in the packets. If the packets are different, it checks the status of the ECU and if the flags are setted to True. Based on the status and the flags it then increments or decrements the ECU's TEC and REC, adding the TEC to an array that stores the its history. Finally, the *send_packet()* function sends, and if necessary, retransmits a packet storing it onto the CAN bus packet_log array. The particularity of the function is that if the ECU that wants to send the packet is the adversary's one, it sends a packet with the control paramaeter equal to 0 (as specified in 3.1)

3.3 CAN bus class

```
# Class representing a simplified CAN bus
class CANBus:
    def __init__(self):…

    def connect_ecu(self, ecu):…

    def disconnect_ecu(self, ecu):…
```

Figure 4: CAN bus class and its methods

The CAN Bus class (Figure 4) represents a really simple CAN bus, indeed it creates the communication channel that stores the ECUs connected to it and the packets sent at each interaction.
The two functions of the class allow an ECU to be connected to the CAN bus or disconnected from it, adding or removing the ECU from the array that stores the connected ECUs.

3.4 Simulation function

```
def run_ecu(ecu, packet_id, data, canbus, barrier):
    """
    Simulate the behavior of an ECU, including sending packets and handling errors
    throug ECU functions.

    Parameters:
    - ecu: The ECU object representing the node in the CAN bus.
    - packet_id: The ID of the packet to be sent by the ECU.
    - data: The data payload to be sent in the packet.
    - canbus: The shared CAN bus object connecting all ECUs.
    - barrier: A threading barrier used to synchronize ECUs' actions.
    """
```

Figure 5: Function that simulate the behavior of the ECU connected to the CAN bus

The *run_ecu()* function (Figure 5) simulates the behaviour of an ECU connected to the CAN bus. The function synchronises the ECUs, makes the ECU send packets over the CAN bus, makes the ECU check

if there are duplicate ID packets inside the CAN bus at the same moment, and if there are duplicate packets it makes the ECU handle the error (retransmission, set flags, increment or decrement of TEC and REC, and disconnection of the ECU if it enters in a Bus-Off state).

4 Results and Discussion

The output of the simulator is the evolution of the Bus-Off attack (Figure 6).

```
The ECU 1 has been connected to the CAN bus
The ECU 2 has been connected to the CAN bus
|  [Adversary] | ECU 2 sent packet: ID=100 PACKET=[1, 100, 1, 0, [0, 0, 1], 0, 0, 1] |
|  [Normal   ] | ECU 1 sent packet: ID=100 PACKET=[1, 100, 1, 1, [1, 0, 1], 0, 0, 1] |
Normal    -> tec: 8 rec: 8 activeFlag: True passiveFlag: False ErrorActive
Adversary -> tec: 8 rec: 8 activeFlag: True passiveFlag: False ErrorActive
------------------------------------------------------------
------------------------------------------------------------
Retrasmission | |  [Adversary] | ECU 2 sent packet: ID=100 PACKET=[1, 100, 1, 0, [0, 0, 1], 0, 0, 1] |
Retrasmission | |  [Normal   ] | ECU 1 sent packet: ID=100 PACKET=[1, 100, 1, 1, [1, 0, 1], 0, 0, 1] |
Normal    -> tec: 16 rec: 16 activeFlag: True passiveFlag: False ErrorActive
Adversary -> tec: 16 rec: 16 activeFlag: True passiveFlag: False ErrorActive
------------------------------------------------------------
```

Figure 6: Simulator's output

The ecu2 (advesary ECU) sends an adversary packet against the ecu1 (victim ECU). This causes both ECUs to set the active flag to True and then increment their TEC by 8 until they reach 128 (16 iterations). Once they reach a TEC of 128, they both enter the error passive state and their behaviour changes. In fact, the victim ECU starts to set the passive flag to true and to increment its TEC by 8, while the adversary ECU decrements its TEC by 1 because the victim ECU has a passive flag set to true (i.e. the equivalent of sending 6 non-dominant bits). From this point on, the victim ECU's TEC increments by 7 (8-1) until it reaches 255, i.e. the threshold of the bus-off state, while the adversary continues to decrement its TEC by 1. As soon as the victim ECU enters the bus-off state, the simulator stops running and prints the graph (Figure 7) of the history of the TEC of both ECU.

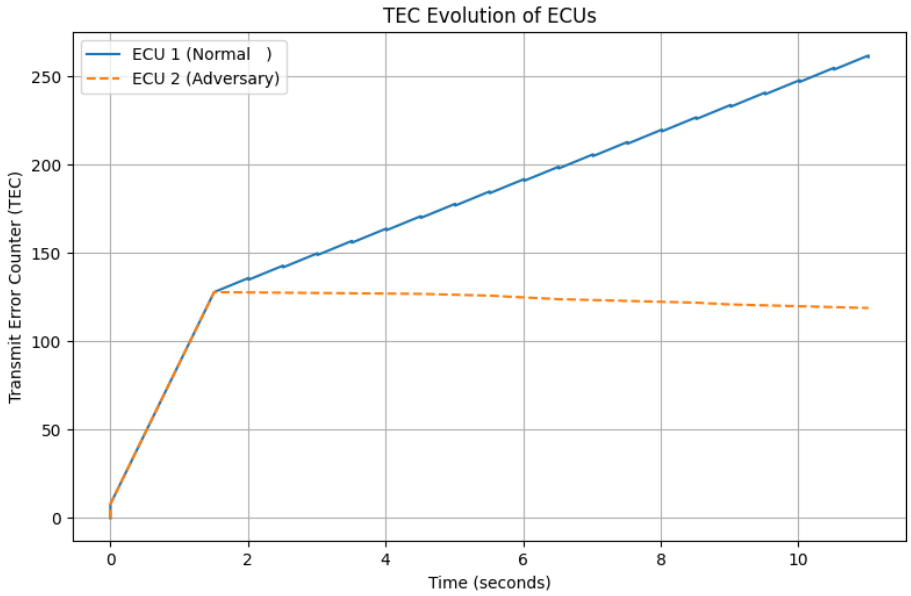

Figure 7: TEC history after the bus-off attack

The graph in Figure 7 is quite similar to the one in the reference paper. The main differences are:

- In the above graph, the TEC starts at 0 but immediately becomes 8. This is due to the fact that the simulator sends an adversary packet at the first iteration and does not simulate the listening phase of an adversary. In fact, in a real scenario, the adversary doesn't know anything about the packet's periodic transmission time, or its ID, or the sending ECU, so he has to look at the CAN bus and understand all these things. Once he understands when to send the packet, with what ID and to which ECU, he creates the packet and then sends it over the CAN bus. The simulator doesn't implement this phase, so the beginning of the graph is a bit different, but still logically correct, because the first packet is a duplicate with a dominant bit difference that leads to a TEC incrementation of both ECUs.
- The graph above shows a slower incrementation of the TEC during the first phase, whereas the TEC graph in the reference paper shows a faster incrementation, with the TEC increasing almost vertically during the first phase (Error Active) and decreasing its incrementation speed during the second phase (Error Passive). This difference is probably due to the fact that in the first phase the simulator doesn't process the recessive bits before a retransmission, and in the second phase it simulates the delay using only a time value.

4.1 Limitations

Even if the simulator disconnects the victim ECU when it reaches the bus-off state after the attack, ensuring that its TEC is higher than 255, the adversary ECU sometimes doesn't decrease its TEC when the victim ECU is in Error Passive state. This can result in the victim ECU incrementing its TEC by 7 (from 8 to 1), while the adversary ECU remains at the same TEC. This issue is likely caused by the fact that the ECUs are managed by two different threads that run concurrently. Sometimes, one thread executes its functions before the other, causing the flags to be set after the check of the other thread, which can bypass the controls for TEC incrementation.
This is a problem related to the simulator's implementation. It provides a simplified representation of the in-vehicle network and does not model the sending of packets over the CAN bus bit by bit as in a real-world scenario. The simulator also does not transmit real error flags (6 bits) when an error occurs, nor does it handle the recessive bit before a retransmission.