Università degli Studi di Padova

Università degli Studi di Padova

Master Degree in Cybersecurity

Report fot the Course Cyber-Physical Systems and IoT Security
Paper number 3, topic: CAN Bus

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

Report and Project:
Nicola Busato 2119291

Academic year 2024/2025

Contents

1 Objectives

The primary goal of this project is to simulate a Bus-Off attack on a Controller Area Network (CAN) system.
The objectives can be divided into the following sub-tasks:

- **Simulator Development**: Create a simulator capable of reproduce a simplified CAN bus system, including Electronic Control Units (ECUs) that send messages over the bus.
- **Attack Simulation**: Implement and execute a Bus-Off attack in the simulator.

The ultimate objective is to generate a plot comparable to the "*Figure 12: TECs during a bus-off attack*" from the reference paper.

2 System Setup

The simulator for this project was developed by me, using Python 3.12.3. The following libraries were used:

- **Time**: For managing timing in message transmission and delays.
- **Threading**: To simulate the concurrent behavior of multiple ECUs on the CAN bus.
- **random**: For introducing randomness in frame generation.
- **matplotlib.pyplot**: For visualizing data, such as TEC values, at the end of the experiments.

2.1 Design Choices

The initial approach was to use a shared variable provided by the CAN bus, where the ECUs could send a Frame Object. However, since the frame transmission by the ECUs can change, send error flags, or even stop transmission if the bus is used by a frame with a lower ID, I decided that a better design choice would be to use bit-level communication with multiple threads, one for the CAN bus and another for the ECUs. The main challenge was synchronizing the operations of the CAN bus and the ECUs.

2.2 Simulator's code

The simulator consists of five Python files:

- **global_clock.py**: Contains the `GlobalClock` class, responsible for synchronising processes with periodic signals.
- **frame.py**: Defines the `Frame`, which emulates a real CAN frame structure with fields such as SOF, ID, DLC, Data, and EOF. For simplicity, CRC and ARK are not included.
- **can_bus.py**: Contain `CanBus` class that simulates a CAN bus communication system.
- **ecu.py**: Contains the `ECU` class, represents an Electronic Control Unit
- **main.py**: The core script of the simulator, initializes the GlobalClock and CAN bus, starts ECU threads and visualizes TEC values over time once the simulation completes.

2.2.1 GlobalClock

The `GlobalClock` class is used to synchronies processes with periodic signals, ensuring that different tasks in the simulation are executed at fixed time intervals.

2.2.2 Frame

The `Frame` class defines the structure of a CAN frame, which includes:

- 1 bit for Start of frame (SOF)
- 11 bits of ID
- Data Length Code (DLC): number of data bytes, 0-8
- The Data: list of data bytes (length must match DLC)
- 7 regressive bits End of frame (EOF)

I decide to not implement Cyclic Redundancy Check (CRC) and Acknowledgement (ARK) for simplicity.
This class contains the methods:

- `getBits`: Returns a list of bits, including stuffed bits.
- `fromBits(bits : list)` Converts a list of bits in a frame, also removing the stuff bits, and returns an Frame object.

2.2.3 CAN bus

The `CANBus` class has two main methods:

- `transmitBit(bit)`: This method is called by the ECUs when they have to send a bit of the frame.
- `process()`: This method stores the value of the bits based on the bitwise operation of the bits from different ECUs. The resulting stored bits can then be read by the ECUs to compare what they sent with what was received on the bus.

The `process()` method is called by the `canBusThread` of `main.my`, after some time slots, in sync with the `GlobalClock`.
The `CANBus` operates in three states:

- **IDLE**: The bus is waiting for the start of a new frame
- **WAIT**: The bus is waiting for the next bit.
- **ACTIVE**: Transmission is currently active on the bus (i.e., a recent bit has been transmitted).

When the CAN bus is in `IDLE` or `WAIT` and an ECU call `transmitBit(bit)`, the CAN bus recive a bit and switches to `ACTIVE` status. After the transmission, when `process()` is called, the bus pass to `WAIT`. If two consecutive `process()` invoke without intermediate state change, i.e. the can bus remain in `WAIT` for two consecutive cyclics, the Can bus switches to `IDLE`. At this point, the ECUs can start to transmits a new frame. The term frame slot is a concept I defined for this simulation. It refers to the specific time interval during which the CAN bus is in a `IDLE` state and is ready to accept a new frame from an ECU.

2.2.4 ECU

The `ECU` class represents an Electronic Control Unit that transmits frames on the CAN bus.
By using the method `sendFrame(frame: 'Frame')`, an ECU starts a bit-level transmission. After transmitting a bit, the ECU checks what the CAN bus received. In case of discrepancies, it raises one of three types of errors, and stop the normal frame transmission:

- `LOWER_FRAME_ID`: The transmitted frame has a higher ID than another frame being transmitted simultaneously.
- `BIT_ERROR`: The ID was transmitted correctly, but there is a discrepancy in following fields of the frame.
- `STUFF_ERROR`: The ECU detects six consecutive bits of the same polarity.

If the transmission is successful, the method returns the string `COMPLETED`
In case of `BIT_ERROR` or `STUFF_ERROR` during the transmission, the ECU stops transmission and begins sending the `ERROR_ACTIVE_FLAG` or the `ERROR_PASSIVE_FLAG`, i.e. six consecutive dominant bits (0) or six consecutive recessive bits (1), depending on the ECU's status `ERROR_ACTIVE` or `ERROR_PASSIVE`. When an error is found the `TEC` value increases by 8, and the ECU do a retransmission. When the transmission completes successfully, the TEC value decreases by 1. The TEC values are saved whenever they increase or decrease.
The ECU switches between the following three states based on the TEC and REC values: `ERROR_ACTIVE`, `ERROR_PASSIVE` and `BUS_OFF`

2.2.5 Main

Creates the `GlobalClock` and starts the `canBus_thread`, which runs a loop that calls `CanBus.process()` and waits three Clock cycles, in this way the ECUs can connect to the CAN bus and sand the bits in a fixed time intervals.
Then, the main module creates the `ecuThread` for each ECU, which handles transmission and retransmission of frames on the CAN bus. When an `ecuThread` is created, it requires a `period` parameter, which represents how often the ECU will attempt to send a frame on the CAN bus. The period is based on the transmission of a frame on the CAN bus; after a transmission ends, the CAN bus enters the `IDLE` state, and the period is counted as the number of `IDLE` states before the ECU can attempt to send its frame. During the transmission, if the ECU misses its scheduled transmission time (e.g., it should transmit at t = 5 but does so at t = 6), the next attempt will be made at t = 10, not at t = lastTransmit + period. In the case of retransmission, the ECU transmits at t = lastSend + 1. The retransmission occurs if the frame transmission returns a status other than `COMPLETED`.
The attacker has its own thread, and the it's behavior will be described in the next chapter.
Once all threads have finished, the main module calls `plot_graph` to visualize the TEC values over time for the ECUs.

2.3 Synchronisation

The initial design relied only on `clock.wait()` from the `GlobalClock`. However, due to hardware limitations, this approach proved insufficient to maintain consistent operation. To address this, I used additional synchronisation techniques:

- `threading.Event()`: Applied in methods such as `waitIdleStatus()` and `waitFrameCount()`.
- `threading.Barrier(n)`: Utilised for barriers like `start_barrier` and `sync_barrier`.

While these methods are not fully representative of real-world CAN bus synchronisation, they were necessary to overcome the sync issues. I tried to avoid introducing such elements, but it was needed to ensure the simulator worked properly and achieved its goals.

2.4 Starting the Simulator

To run the simulator, execute the following command:

```
1 python3 main.py
```

Listing 1: Shell command for start the simulator.

2.4.1 Simulation Configuration

It is possible to adjust certain parameters of the simulator:

- **CLOCK**: Time step in seconds. Recommended value is 0.003 or higher to avoid unpredictable behavior.
- **ECU_NUMBER**: Number of additional ECUs (excluding the Victim and Adversary). **Note**: The program allows the creation of extra ECUs, but in the current implementation have synchronisation issues with multiple ECUs.
- **PERIOD**: Transmission period for the Victim ECU. This value must be at least 5 to ensure proper synchronisation. I introduced a wait between the transmission slots; therefore, the higher the `PERIOD` value, the longer the simulation takes. However, a higher `PERIOD` results in a final plot that is more similar to the one presented in the reference paper. In the repository's "Other" folder, you can find plots with different `PERIOD` values: 5, 10, and 20.

3 Experiments

In this section, I describe the methodology used to simulate the attack scenario.

3.1 Attack structure

The attack is structured as follows:

1. **Attacker Simulation**: To simulate the attacker, I implemented the method `attackerThread(canBus)`. This method listens for frames on the CAN bus and identifies the first frame as the "Victim". Once the Victim is identified, the attacker waits for the next frame from the same ECU, which allows it to calculate the transmission period of the Victim ECU.

2. **Frame Forging**: After determining the Victim's period, the attacker creates a new ECU and transmits a frame with the same ID and period as the Victim's. The adversarial frame is specifically designed to have the same ID as the Victim but with a DLC of 0, meaning it contains no data, this is an unusual configuration for CAN frames, which increases the likelihood of causing bit errors during transmission.

3. **Attack Execution**: When the Victim and Adversary share the same transmission period, their transmissions collide, causing discrepancies in the transmitted bits. This increse the TEC of both for some period and then only the Victim TEC increses, better explenation in next chapter

The attack code is implemented in the method `attacker(canBus)`, which is called by the `main` function.

The full code can be found in the https://github.com/Nicola-01/CPS-IoT/tree/main/project1

4 Results and Discussion

In this section, I present the results obtained from the simulator and discuss the logs, comparisons with the reference paper, and possible improvements for future simulations.

4.1 Simulators logs

4.1.1 Log structure

After starting the simulation, the logs provide detailed updates about the simulator's behavior:

```
1  Start Victim -> Period: 5 ; Frame(SOF=0, ID=1349, DLC=1, Data=[8], EOF=127)
2     Victim    | ECU: TEC: 0  , Status: ERROR_ACTIVE  | Transmitted frame status: COMPLITED | CanBus slot: 0
3     Victim    | ECU: TEC: 0  , Status: ERROR_ACTIVE  | Transmitted frame status: COMPLITED | CanBus slot: 5
4  Adversary found Victim's period: 5
5  Start Adversary -> Period: 5 ; Frame(SOF=0, ID=1349, DLC=0, Data=[], EOF=127)
6     Victim    | ECU: TEC: 8  , Status: ERROR_ACTIVE  | Transmitted frame status: BIT_ERROR | CanBus slot: 10
7     Adversary | ECU: TEC: 8  , Status: ERROR_ACTIVE  | Transmitted frame status: BIT_ERROR | CanBus slot: 10
8     Victim    | ECU: TEC: 16 , Status: ERROR_ACTIVE  | Transmitted frame status: BIT_ERROR | CanBus slot: 11
9     Adversary | ECU: TEC: 16 , Status: ERROR_ACTIVE  | Transmitted frame status: BIT_ERROR | CanBus slot: 11
10    ...
11    Victim    | ECU: TEC: 120, Status: ERROR_ACTIVE  | Transmitted frame status: BIT_ERROR | CanBus slot: 24
12    Adversary | ECU: TEC: 120, Status: ERROR_ACTIVE  | Transmitted frame status: BIT_ERROR | CanBus slot: 24
13    Victim    | ECU: TEC: 128, Status: ERROR_PASSIVE | Transmitted frame status: BIT_ERROR | CanBus slot: 25
14    Adversary | ECU: TEC: 128, Status: ERROR_PASSIVE | Transmitted frame status: BIT_ERROR | CanBus slot: 25
15    Victim    | ECU: TEC: 136, Status: ERROR_PASSIVE | Transmitted frame status: BIT_ERROR | CanBus slot: 26
16    Adversary | ECU: TEC: 127, Status: ERROR_ACTIVE  | Transmitted frame status: COMPLITED | CanBus slot: 26
17    Victim    | ECU: TEC: 135, Status: ERROR_PASSIVE | Transmitted frame status: COMPLITED | CanBus slot: 27
18    Victim    | ECU: TEC: 143, Status: ERROR_PASSIVE | Transmitted frame status: BIT_ERROR | CanBus slot: 30
19    ...
20    Victim    | ECU: TEC: 248, Status: ERROR_PASSIVE | Transmitted frame status: BIT_ERROR | CanBus slot: 105
21    Adversary | ECU: TEC: 111, Status: ERROR_ACTIVE  | Transmitted frame status: COMPLITED | CanBus slot: 105
22    Victim    | ECU: TEC: 247, Status: ERROR_PASSIVE | Transmitted frame status: COMPLITED | CanBus slot: 106
23    Victim    | ECU: TEC: 255, Status: ERROR_PASSIVE | Transmitted frame status: BIT_ERROR | CanBus slot: 110
24    Adversary | ECU: TEC: 110, Status: ERROR_ACTIVE  | Transmitted frame status: COMPLITED | CanBus slot: 110
25    Victim    | ECU: TEC: 254, Status: ERROR_PASSIVE | Transmitted frame status: COMPLITED | CanBus slot: 111
26    Victim    | ECU: TEC: 262, Status: BUS_OFF       | Transmitted frame status: BIT_ERROR | CanBus slot: 115
27 Victim entered BUS_OFF. Stopping all threads.
28    Adversary | ECU: TEC: 109, Status: ERROR_ACTIVE  | Transmitted frame status: COMPLITED | CanBus slot: 115
29 All threads stopped.
```

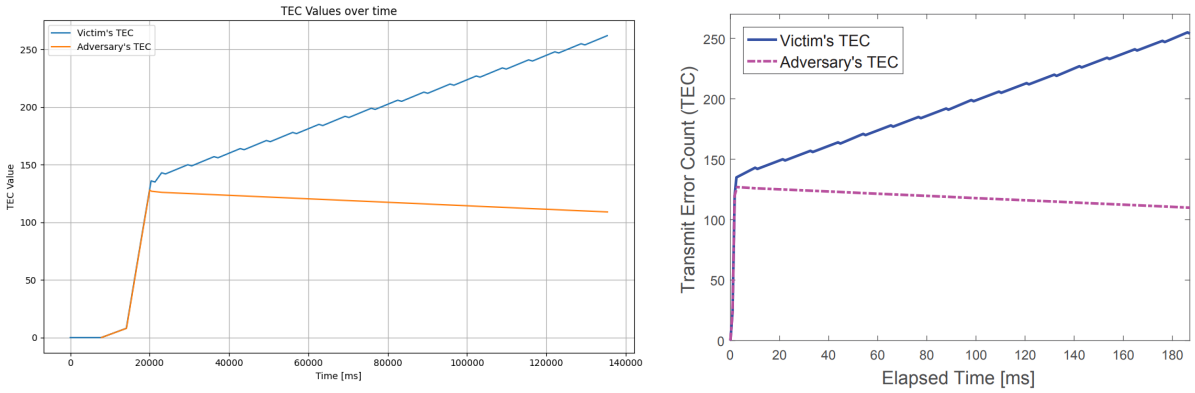
Figure 1: Sample log output during the Bus-off attack. Some logs have been removed.

4.1.2 Log Analysis

- **Start Messages**: Line 1 and 5 show the initialization of an ECU, showing their respective transmission periods and frame contents. For simplicity, the Data field remains static in this simulation. However, it can easily be modified to change dynamically after each completed transmission.
- **Adversary Period Detection**: Line 4 indicates that the Adversary identified the Victim's transmission period.
- **Transmission Logs**: Lines 2-3, 6-26 represent the ECU log format, which includes
  - **ECU Name**: Identifies the ECU, e.g. Victim or Adversary, or other ECUs.
  - **ECU TEC**: Transmit Error Counter value after transmission.
  - **ECU Status**: ECU status after transmission, i.e. `ERROR_ACTIVE`, `ERROR_PASSIVE` or `BUS_OFF`.
  - **Frame Status**: Status of the transmitted frame, i.e., `COMPLETED`, `BIT_ERROR`, `STUFF_ERROR`, or `LOWER_FRAME_ID`.
  - **CanBus Slot**: Indicates the transmission slot used by the ECU.
- **Bus-Off Event**: Line 27 shows the moment when the Victim ECU enters the `BUS_OFF` state, after which the simulator shuts down all threads.

4.2 Plots comparison

I compared the plot generated by my simulator with the one from the paper.



(a) Bus-off attack on my simulator, with PERIOD of 10

(b) Bus-off attack on the paper

Figure 2: Comparison between the plot obtained from my simulator (a) and the plot from the paper (b).

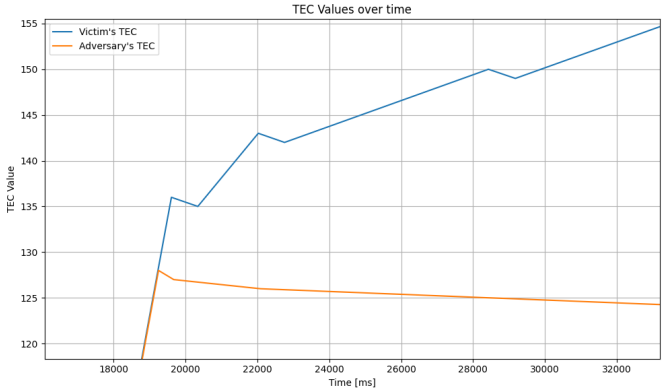


Figure 3: Zoom of the plot showing the transition from Phase 1 to Phase 2

The results from my simulator closely match the reference paper. The main differences in timing are due to simplifications in my simulator's implementation, but the results are comparable. Now I will explain the different sections of the plots and compare them, times refer to my plot.

- **0–8000ms**: The flat line represents the period when the victim ECU is not yet under attack, meaning the adversary has not started the attack yet.
- **8000-15000ms**: At around 15000ms, the adversary starts the attack. Since TEC values are saved only when the transmission ends (either in `COMPLITED` status or an error status), the period around 7000ms represents a transmission period. It is possible to see that also the paper's plot is not a completely straight line and has a slight bend at the start.
- **15000-19500ms**: [Phase 1, according to the paper] In this phase, both the victim and adversary are in `ERROR_ACTIVE`. Since the adversary has a frame containing at least one dominant bit (0), while the victim's bit is recessive (1), the victim detects the discrepancy in a bit between what it sends and what the CAN bus returns. This causes the victim to trigger a `BIT_ERROR` and start transmitting the `ERROR_ACTIVE_FLAG`, i.e., six consecutive bits with the same polarity, all dominant (0). This leads the adversary to detect a `BIT_ERROR` since the `ERROR_ACTIVE_FLAG` overwrite the transmission, when adversary sees the error increases its TEC by 8 and sends the `ERROR_ACTIVE_FLAG`. This behavior matches the paper's plot. In the logs of Figure [1], this corresponds to lines 6 to 9.

- **19500-20500ms**: [Phase 1 to 2, according to the paper] This phase is zoomed in Figure [3]. The highest point of the adversary's TEC is 128. At this peak, both ECUs are in `ERROR_PASSIVE`. The flag raised is the `ERROR_PASSIVE_FLAG`, i.e., six consecutive recessive bits (1). After both reach `ERROR_PASSIVE`, a new retransmission starts. However, since both are in `ERROR_PASSIVE`, when the victim detects the `BIT_ERROR`, it increases the TEC of 8, from 128 to 136, and sends the `ERROR_PASSIVE_FLAG`, which does not overwrite the transmitted bit of the adversary (which has no error), so the adversary completes the transmission with `COMPLETED` status and decreases the TEC of 1, from 128 to 127, returning to `ERROR_ACTIVE` status. In the logs of Figure [1], this corresponds to lines 13 to 16.

  After this, the adversary completes its transmission and must wait for its period to transmit again, so the victim can complete its retransmission with COMPLETED status at 20500ms (log line 17).
- **20500 onwards**: [Phase 2, according to the paper] Both the adversary and the victim start their transmissions according to their periods. The adversary can complete its transmission without encountering any collisions, but the victim detects a collision in one of its bits due to the differing frames, increases its TEC, and sends the `ERROR_PASSIVE_FLAG`. This does not cause any collision since the flag is a recessive bit (1). After the error, the victim waits for retransmission on the next transmission slot, which the adversary cannot use since it must wait for its period. This allows the victim to complete its transmission of the frame. This process continues until the victim's TEC exceeds 255, at which point it enters the `BUS_OFF` status, and the simulation ends.

  Note From 20500ms to 22500ms, this is the first transmission where the Adversary is in `ERROR_ACTIVE` and Victim in `ERROR_PASSIVE`. Since all the previous transmissions were in retransmission, with transition every available transmission slot, it must now wait for the normal time slot based on the period. Therefore, the transmission might start sooner than a normal transmission. For example, if the victim has a period of 10, it transmits at slots 10, 20, 30, etc. If Phase 1 to 2 finishes at slot 25, the normal transmission will start at 30, which is sooner than a normal transmission. This behavior matches the real plot.

4.3 Future improvments

4.3.1 Use of multiple ECUs

The simulator supports multiple ECUs through the `ECU_NUMBER` constant, but synchronization issues cause transmissions to occur in different CAN bus slots. A possible solution is using a barrier, though this doesn't fully reflect real-world behavior.

4.3.2 Remouve threading.Barrier

Due to bugs I couldn't resolve, I had to use `threading.Barrier()` for synchronizing the victim and adversary ECUs. I tried using a custom `waitFrameCount()` function, but even that wasn't an ideal solution and still had synchronization issues.

4.3.3 Better CAN Bus Fidelity

I simplified the CAN bus transmission by introducing the concept of a "*transmission slot*", which works for the simulator. For simplicity, retransmissions occur in the next available transmission slot. However, for retransmissions, some slots should be "*retransmission slots*" if no ECU transmits for a while, for emulate the 11-bit IFS. This feature is partially implemented but not yet fully in use.