Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master in ICT for Internet and Multimedia

Mid-Project Report of The Course CyberPhysical Systems and IoT Security

Title of the reference paper:
Error Handling of In-vehicle Networks Makes Them Vulnerable

Group Members

Surname	Name	Student ID
NEMMICHE	Mohamed Mounib	2184905
SAIDI	Hassiba Lina	2142744
PERIN	Alessandro	2203653

Academic Year
2025/2026

1 Objectives

The main objective of this project is to **simulate a Bus-Off attack on a Controller Area Network (CAN) bus** and reproduce, in a controlled software environment, the behavior described in the reference paper *"Error Handling of In-vehicle Networks Makes Them Vulnerable"* by Cho and Shin [1]. The attack forces a healthy victim ECU into the Bus-Off state by exploiting the CAN error-handling mechanism, thereby silencing it from the network without needing to understand the application-level semantics of messages [1].

More specifically, the project aims to:

- Design and implement a **discrete-time Python simulator** that models a simplified CAN bus with arbitration, collisions, error flags, and Transmit Error Counter (TEC) evolution for each ECU [2].
- Implement a **victim ECU** that periodically transmits CAN frames and an **attacker ECU** that first observes traffic to identify patterns (pattern analysis phase), then injects carefully timed frames to increment the victim's TEC and drive it from Error-Active to Error-Passive, and finally to Bus-Off [2].
- Reproduce the characteristic **Phase 1 and Phase 2 TEC patterns**: Phase 1 with both victim and attacker TECs increasing, and Phase 2 with the victim's TEC increasing in a stepwise fashion while the attacker's TEC decreases or stabilizes, as reported in the reference paper [1, 2].
- Evaluate how **CAN bus speed** (250, 500, 1000 kbps) affects the time needed to reach Error-Passive and Bus-Off states, using multiple independent attack trials and statistical analysis (box plots, success rates) [2].

In addition to the simulator, the project includes a **practical demonstration using the ICSim Instrument Cluster Simulator** to show how CAN frames can be sniffed and injected on a virtual `vcan0` interface, making visible the impact of malicious traffic on a vehicle's dashboard [3]. This complements the Bus-Off simulation by illustrating how an attacker can both disrupt and manipulate in-vehicle communications.

2 System Setup & Implementation

This section describes the architecture of the simulation framework, the main Python modules, and how they interact to model the Bus-Off attack scenario. The implementation is entirely software-based, using Python 3 and standard scientific libraries for logging and analysis.

2.1 Simulation Overview

The simulation models a **single CAN bus** shared by two primary nodes:

- A **Victim ECU**, which periodically transmits CAN frames with a fixed ID (and, optionally, preceded frames) and maintains TEC and state according to the CAN fault-confinement rules [2].
- An **Attacker ECU**, which initially observes bus traffic to detect the victim's periodic behavior (pattern analysis), then synchronizes its own transmissions to cause collisions and escalate the victim's TEC until Bus-Off [2].

The behavior of the bus, ECUs, and the attack is implemented in the `Simulation/` folder, while experiment results are logged as **JSON Lines** in `Simulation/attack_logs/` and later analyzed via the `Notebook/attack_graphPlots.ipynb` notebook [2].

2.2 Repository Structure

The core simulation is organized as follows:

- `Simulation/main.py`: Entry point that configures and runs both single-run and multi-run experiments, and writes log files [2].
- `Simulation/can_bus.py`: Implements the CAN bus model, including arbitration, collision detection, and error flag handling.
- `Simulation/ecu.py`: Defines the base ECU class, encapsulating TEC, error state (Error-Active, Error-Passive, Bus-Off), and the update of counters based on transmission outcomes.
- `Simulation/victim_ecu.py`: Extends the base ECU to implement the victim's periodic traffic pattern (and, if needed, preceded/non-periodic frames).
- `Simulation/attacker_ecu.py`: Extends the base ECU to implement the attacker, including pattern analysis of victim traffic and timing of attack frames.
- `Simulation/setup_logger.py`: Provides utilities to log simulation events to JSON Lines files with a consistent schema for later analysis.
- `Simulation/attack_logs/`: Folder where the simulator stores `single_run.log` and the aggregated logs (e.g., `attack_250kbps.log`).
- `Notebook/attack_graphPlots.ipynb`: Notebook that reads logs, computes metrics (e.g., time to Error-Passive / Bus-Off), and generates the plots used in the discussion.

2.3 CAN Bus Model

The `CANBus` class (found in `can_bus.py`) represents the shared bus and provides the communication medium between ECUs. It implements:

- **Frame scheduling**: ECUs call a method such as `send_frame(frame)` to request transmission in the next simulation step. The bus maintains a list of current transmission attempts.
- **Arbitration**: If multiple ECUs attempt to transmit during the same step, the bus selects the frame with the **lowest ID** as the winner, reflecting the CAN priority rule [1].
- **Collision detection and error handling**: When two frames with the same ID but different contents are transmitted simultaneously, the bus treats this as a **bit error** scenario for the losing ECU(s), which results in TEC increments and triggers error flags [1].
- **Callbacks to ECUs**: After resolving each step, the bus notifies ECUs of the result (e.g., `success`, `bit_error`), so they can update their TEC and state.

2.4 ECU Base Class

The `ECU` base class (found in `ecu.py`) models error handling and state transitions as defined in the CAN specification and exploited by the Bus-Off attack:

- **Transmit Error Counter (TEC)**: Incremented by a configured amount (e.g., +8) on each transmit error and decremented (e.g., -1) on successful transmissions [1].
- **Error states**:
  - **Error-Active**: Default state where the ECU can send active error flags and participate normally in the bus.
  - **Error-Passive**: Reached when TEC ≥ 128, where the ECU sends passive error flags and its errors no longer disrupt other nodes [1].

  - **Bus-Off**: Reached when TEC ≥ 256, at which point the ECU is disconnected from the bus and stops transmitting [1].

- **State transitions**: The class updates the ECU's state based on TEC thresholds and invokes logging hooks to record each change.

2.5 Victim ECU

The `VictimECU` class (found in `victim_ecu.py`) represents the target of the attack:

- It periodically sends CAN frames with a fixed ID and data payload, as configured in `main.py`.
- Optionally, it can send **preceded frames**: lower-priority frames that precede the periodic frame at predictable intervals, enabling the attacker to leverage the preceded-ID technique described in the reference paper [1].
- The victim does not attempt to detect or defend against the attack; it is a "normal" ECU that simply reacts to errors by updating its TEC and state, eventually entering Bus-Off.

2.6 Attacker ECU

The `AttackerECU` class (found in `attacker_ecu.py`) models a compromised ECU with the ability to observe and inject frames:

- **Pattern Analysis Phase**: Initially, the attacker passively listens to the CAN bus to detect the victim's periodic frames and, if enabled, their preceding IDs. It computes the period and determines the right timing to collide with the victim's transmissions [2].
- **Attack Phase 1**: Once the pattern is known, the attacker transmits frames with the same ID as the victim at the same time, but with deliberately different contents, causing **bit errors** and increasing both TECs until both reach the Error-Passive threshold.
- **Attack Phase 2**: After the victim becomes Error-Passive, its passive error flags no longer disrupt the attacker, so the attacker can often transmit successfully while the victim continues to accumulate errors and move toward Bus-Off, reproducing the Phase-2 "sawtooth" TEC pattern [1].

2.7 Logging and Analysis

The `setup_logger.py` module writes simulation events to JSON Lines files:

- **Single-run logs** (`single_run.log`) record a fine-grained timeline of one representative attack (TEC values, states, and phase labels at each time step).
- **Aggregated logs** (e.g., `attack_500kbps.log`) store per-trial statistics (time to Error-Passive, time to Bus-Off, final TEC, success flag) for 1000 runs at each speed.

The notebook `attack_graphPlots.ipynb` then reads these logs, reconstructs TEC trajectories, and generates the plots (TEC vs time, box plots by speed, success/failure bar charts) used in the Results and Discussion section [2].

3 Experiments

This section details the experimental methodology used to validate the Bus-Off attack simulation. The experiments were divided into two main categories: verifying the attack mechanics through detailed single-run analysis, and evaluating the impact of bus parameters through statistical batch processing.

3.1 Experimental Methodology

To reproduce the results of the reference paper [1], the simulation was configured to run a "Periodic Attack" scenario. In this scenario, the Victim ECU transmits a frame with a specific ID every 10 milliseconds. The Attacker ECU is configured to target this specific ID.

The experiments were conducted in three distinct phases:

1. **Pattern Analysis Validation**: The first step verified that the Attacker ECU could correctly identify the victim's transmission period. The attacker was set to listen for 10 periods (approx. 100ms) before initiating any collisions.

2. **Phase 1 & 2 Transition**: We observed the evolution of the Transmit Error Counters (TEC) to confirm the transition from *Error-Active* to *Error-Passive*. This is critical because the attack strategy changes once the victim enters the passive state (where its error flags no longer dominate the bus) [1].

3. **Speed Dependency Analysis**: We varied the CAN bus speed across three standard automotive bit rates—250 kbps, 500 kbps, and 1000 kbps—to measure how the physical speed of the bus correlates with the time required to silence the victim.

3.2 Simulation Parameters

All experiments used the discrete-time simulator described in Section 2. The key configuration parameters were:

- **Victim Period**: 10 ms
- **TEC Increment (Error)**: +8 (Standard CAN specification)
- **TEC Decrement (Success)**: -1 (Standard CAN specification)
- **Bus Speeds**: 250 kbps, 500 kbps, 1000 kbps
- **Time Step**: Dynamic, calculated based on the bit rate (e.g., lower bit rates result in longer frame durations).

3.3 Data Collection

Data was collected using the custom logging system implemented in `setup_logger.py`, which generates four distinct log files in the `attack_logs/` directory [2]:

- `single_run.log`: This file contains a high-resolution trace of a single representative attack at 500 kbps. It records the state of every ECU at every simulation step, including timestamped values for the Victim's TEC, the Attacker's TEC, and the current bus phase (Analysis, Phase 1, Phase 2). This log is used to visualize the "Sawtooth" pattern and the state transitions.

- **`attack_250kbps.log`, `attack_500kbps.log`, and `attack_1000kbps.log`**: These three files store the results of the batch experiments. Each file contains 1,000 entries (one per trial) for its respective bus speed. They log aggregate metrics rather than time steps:
  - *Time to Error-Passive*: The duration from the start of the attack until the victim's TEC reached 128.
  - *Time to Bus-Off*: The total duration until the victim's TEC reached 256.
  - *Success Indicator*: A boolean flag confirming if the victim successfully reached the Bus-Off state.

These logs were processed using the `attack_graphPlots.ipynb` notebook to generate the visual results discussed in the following section.

4 Results and Discussion

This section presents the results obtained from the Python simulation, analyzing both the microscopic behavior of the attack (TEC evolution) and the macroscopic impact of bus parameters (speed vs. attack duration).

4.1 TEC Evolution and Attack Phases

The first analysis focused on validating the attack logic by observing the Transmit Error Counters (TEC) during a single attack at 500 kbps. The results are illustrated in Fig. 1 and Fig. 2.

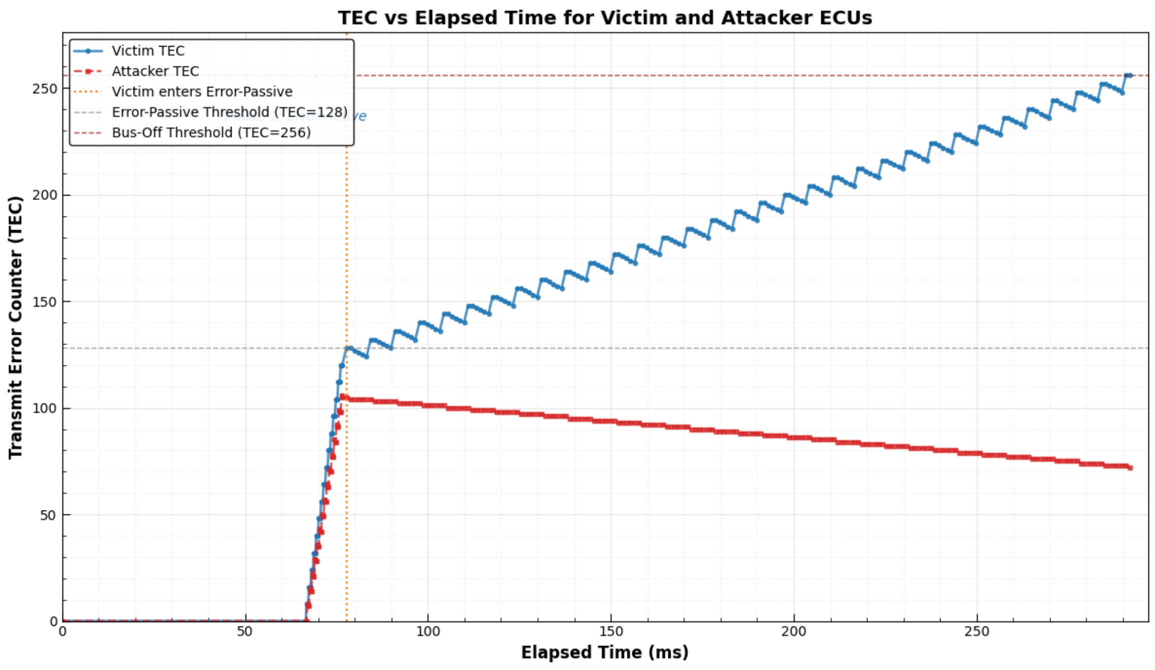

Figure 1: Evolution of Victim and Attacker TEC over time. The vertical dotted line marks the transition to the Error-Passive state.

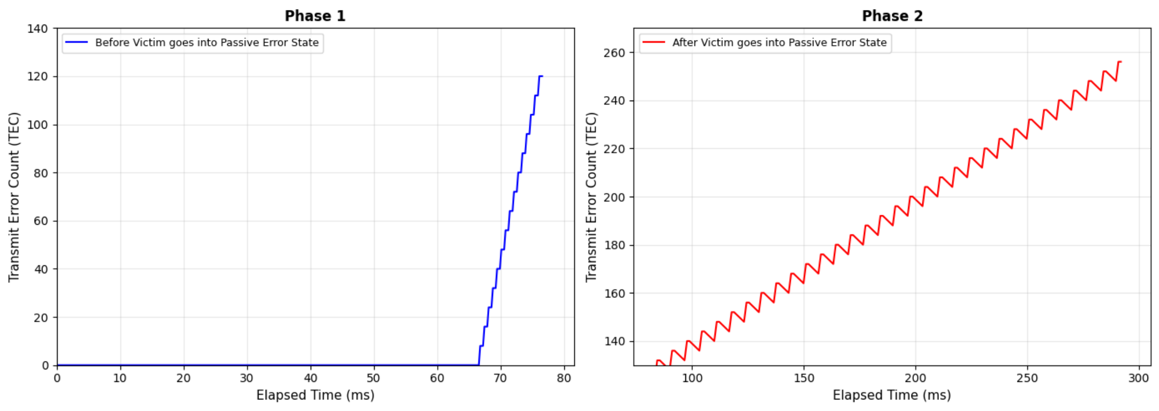

Figure 2: Detailed view of Phase 1 (Staircase pattern) vs. Phase 2 (Sawtooth pattern).

- **Pattern Analysis (0 – 66.6 ms)**: During the initial ≈66 ms, the Attacker's TEC remains at 0. This confirms the "Pattern Analysis" phase where the attacker passively monitors the bus to learn the victim's 10ms period without transmitting.
- **Phase 1 - The Staircase (66.6 – 77.7 ms)**: Once synchronized, the attacker begins injecting collisions. As shown in Phase 1 (Fig. 2), the Victim's TEC rises in sharp increments of +8 (from

0 to 128). This "staircase" pattern confirms that the attacker is successfully causing bit errors on every attempt.

- **Phase 2 - The Sawtooth (77.7 – 290.8 ms)**: At approximately 77.7 ms, the Victim's TEC crosses the 128 threshold, entering the *Error-Passive* state. From this point, the Victim's TEC exhibits a "sawtooth" pattern (rising by +8 on collision, then decreasing by -1 on successful retransmissions), while the Attacker's TEC (red line in Fig. 1) monotonically decreases. This confirms the core vulnerability: once the victim is passive, its errors are internal, allowing the attacker to recover while pushing the victim further toward Bus-Off [1].
- **Bus-Off**: The attack concludes at ≈291 ms when the Victim's TEC hits 256, effectively silencing the node.

4.2 Impact of CAN Bus Speed

To understand how physical layer parameters affect attack efficiency, we performed 1,000 Monte Carlo trials at 250 kbps, 500 kbps, and 1000 kbps. The results are summarized in Fig. 3 and Fig. 4.

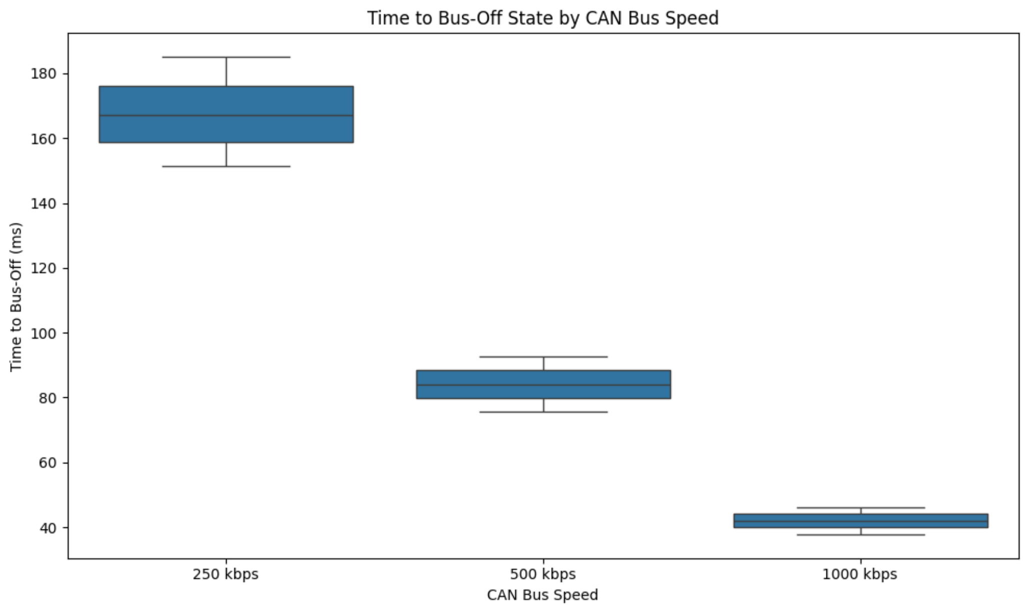

Figure 3: Box plot showing the Time to Bus-Off state across different bus speeds.

- **Time to Bus-Off**: There is a clear inverse relationship between bus speed and attack duration.
  - **250 kbps**: Average time to Bus-Off was **167.65 ms**.
  - **500 kbps**: Average time to Bus-Off dropped to **84.15 ms**.
  - **1000 kbps**: The attack was fastest, silencing the victim in just **42.07 ms**.

  This trend occurs because higher bit rates shorten the frame duration, allowing more attack cycles (collisions) to occur within the same absolute time window [2].
- **Time to Error-Passive**: A similar trend is observed in Fig. 4, with the transition to the passive state occurring in as little as **4.06 ms** at 1000 kbps, compared to **16.24 ms** at 250 kbps. This indicates that at high speeds, a victim has almost no time to recover or detect the anomaly before entering a degraded state.

4.3 Success Rate and Reliability

Fig. 5 illustrates the reliability of the attack across the tested speeds.

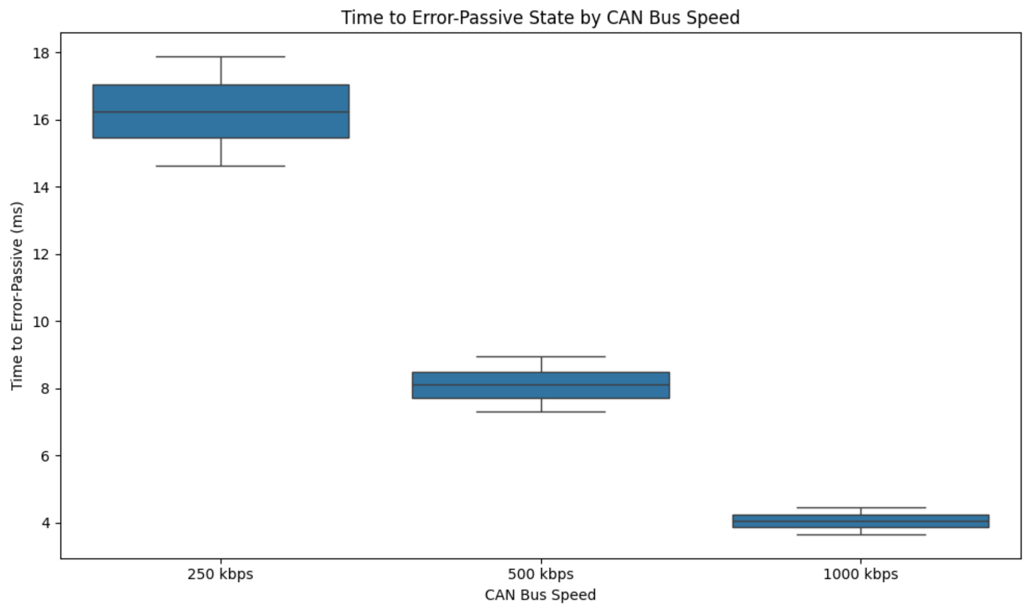

Figure 4: Box plot showing the Time to Error-Passive state across different bus speeds.

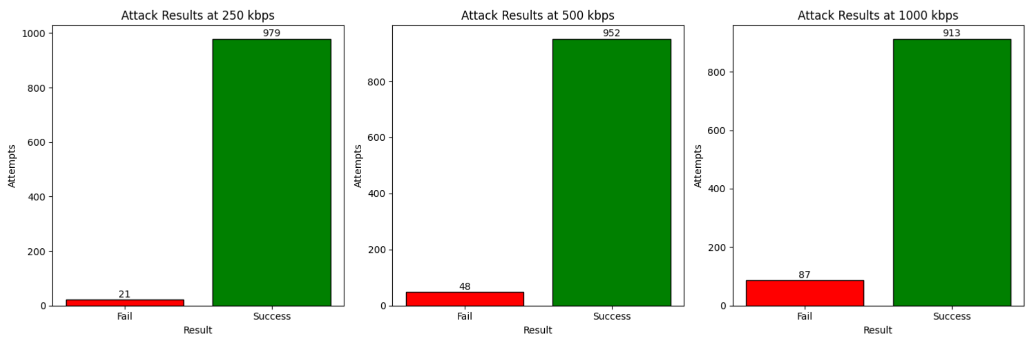

Figure 5: Success vs. Failure rates for 1000 trials at each bus speed.

- **High Reliability**: The attack proved highly effective, with success rates exceeding 91% in all scenarios.
  - 250 kbps: **97.9%** success.
  - 500 kbps: **95.2%** success.
  - 1000 kbps: **91.3%** success.
- **Speed vs. Stability Trade-off**: While higher speeds enable faster attacks, they also introduce a slight decrease in reliability (from $\approx$98% to $\approx$91%). This is likely due to the tighter timing margins at 1000 kbps, where any minor synchronization jitter in the simulator (or real hardware) can cause the attacker to miss the victim's frame, allowing the victim to transmit successfully and decrement its TEC.

4.4 Limitations

While the simulation successfully reproduces the behavior described in the reference paper [1], a few limitations should be noted:

1. **Absolute Timing**: The specific timing values (e.g., 42 ms vs 167 ms) are dependent on the simulation's discrete time steps and may differ slightly from a physical CAN bus with propagation delays.

2. **Simplified Arbitration**: The simulation abstracts bit-level arbitration, assuming immediate collision detection, whereas real controllers have specific sampling points.
3. **Jitter Modeling**: The random jitter used to model real-world variance is an approximation; actual hardware may exhibit more deterministic phase drifts.

Despite these abstractions, the qualitative behavior—specifically the TEC sawtooth pattern and the speed dependency—aligns strongly with the theoretical model.

5 PART$_2$: $PracticalDemonstration : ICSimExploitation$

While the previous sections focused on the theoretical simulation of a Bus-Off attack, this section presents a practical exploitation scenario using the **Instrument Cluster Simulator (ICSim)** [3]. The goal of this experiment was to demonstrate how an attacker with access to the CAN bus can reverse-engineer traffic and inject malicious frames to physically manipulate vehicle behavior (in this case, disabling the accelerator).

5.1 Experimental Setup

The experiment was conducted in a virtualized Linux environment using the SocketCAN framework. The setup consisted of three main components [3]:

- **vcan0 Interface**: A virtual CAN network interface created to route traffic between the simulator and the attack tools.
- **ICSim Dashboard**: A graphical application simulating a vehicle's instrument cluster (speedometer, turn signals, door locks).
- **ICSim Controls**: A controller application allowing user input (e.g., accelerating with the Up Arrow key) to generate legitimate CAN traffic on the bus.

5.2 Reverse Engineering with CANSniffer

The first step was to identify the CAN ID responsible for the acceleration signal. We used the `cansniffer` utility to filter and visualize dynamic traffic on the `vcan0` interface.

By filtering out static IDs and observing the changes when the Up Arrow key (acceleration) was pressed, we identified that the CAN ID `0x19B` was the critical control ID related to the accelerator/engine state.

5.3 Attack Execution: Accelerator Denial-of-Service

Once the target ID (`0x19B`) was identified, we developed a Python script using the `socket` library to perform a Denial-of-Service (DoS) attack on the accelerator function.

The attack logic, shown in Listing 1, operates by continually sending high-priority frames containing a payload of all zeros (`00 00 00 00 00 00 00 00`). This effectively overwrites the legitimate accelerator values, forcing the vehicle to interpret the throttle position as "0" regardless of the driver's input.

```python
import socket
import struct
import sys

TARGET_ID = 0x19B
DATA = b'\x00\x00\x00\x00\x00\x00\x00\x00'  # Zero payload (throttle = 0)
INTERFACE = "vcan0"

def main():
    print("STARTING ATTACK")
    try:
        # Create a raw CAN socket
        sock = socket.socket(socket.PF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
        sock.bind((INTERFACE,))
    except OSError as e:
        print(f"Can't connect to {INTERFACE}. {e}")
        sys.exit(1)

    # Pack the CAN frame (ID, DLC, Data)
    can_fmt = "<IB3x8s"
    can_frame = struct.pack(can_fmt, TARGET_ID, len(DATA), DATA)

    try:
        while True:
            sock.send(can_frame)  # Flood the bus
    except KeyboardInterrupt:
        print("\n attack stopped")
        sock.close()
        sys.exit(0)
    except Exception as e:
        print(f"\n Error during send: {e}")

if __name__ == "__main__":
    main()
```

Listing 1: Python script for Accelerator DoS Attack

5.4 Results

Upon executing the script:

1. The script successfully bound to the `vcan0` interface.
2. The bus was flooded with frames having ID `0x19B` and a zero payload.
3. **Impact**: The speedometer on the ICSim dashboard immediately dropped to 0 km/h, and the vehicle became unresponsive to the Up Arrow key. This confirms that the injected frames successfully overrode the legitimate control signals, effectively creating a denial-of-service condition for the accelerator.

Video Demonstration: A full recording of this attack, showing the setup, execution, and impact on the dashboard, can be viewed here:
`https://drive.google.com/file/d/1amfYHLytMliT9DlFea8LNo9iZGy5vYAD/view?usp=sharing`

This demonstration highlights a critical lack of authentication in standard CAN implementations: any node (or compromised entry point) can spoof critical control messages once the correct ID is identified.

Group Members Contributions

- **Mohamed Mounib NEMMICHE**: I was responsible for the code simulation. This included developing the core logic for the Bus-Off attack, running the experimental trials (single-run and batch), and generating the data plots used for the analysis in Section 4.
- **Hassiba Lina SAIDI**: I was responsible for the documentation. This involved analyzing the simulation data, synthesizing the theoretical concepts from the reference paper, and structuring the final report.
- **Alessandro PERIN**: I was responsible for the practical demonstration. This included setting up the ICSim environment, performing the reverse engineering of CAN frames, and executing the practical Denial-of-Service attack on the instrument cluster.

References

[1] 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*, 2016.

[2] “Source code link of the midproject - cyberphysical systems and iot security,” https://github.com/MounibNemmiche/MidProject-CP-IoTSec.

[3] “Icsim: Instrument cluster simulator,” https://github.com/zombieCraig/ICSim, open Source Tool.