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

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

Name of who is involved in the project: Michael Amista' (2122865), Marco Brigo (2121727)

Source code: https://github.com/MikeIsBack/CPS-Mid-term-Project

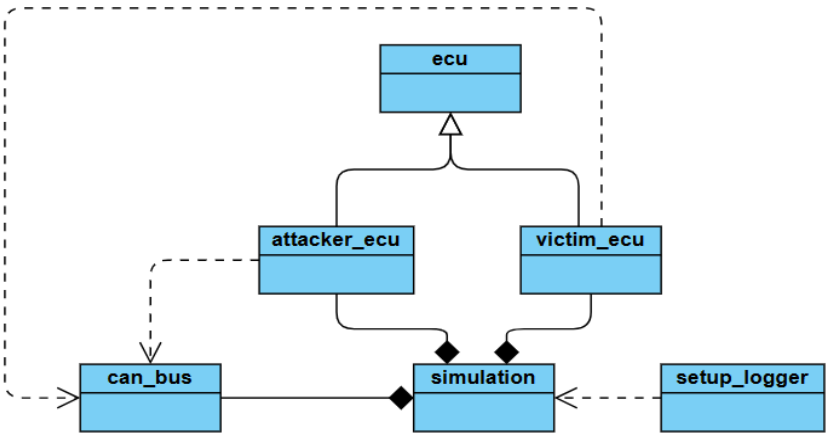
Objectives

The primary goal of this project is to simulate a Bus-Off Attack within a CAN bus network, demonstrating how a compromised ECU (Electronic Control Unit) can exploit inherent vulnerabilities in the protocol to isolate a victim ECU from the network. In this context, the attacker manipulates the error-handling mechanism of the CAN bus to render the victim ECU incapable of transmitting any further messages. The project involves the development of a simulation framework to achieve several key objectives.

First, the simulation must model a simple CAN bus network consisting of two ECUs: a victim ECU, which represents the target of the attack, and an attacker ECU, which has been compromised to execute the malicious behaviour. Second, the attacker ECU is programmed to monitor the traffic on the CAN bus, identify patterns in the victim ECU's transmissions, and plan an attack accordingly. The attacker uses this information to inject malicious frames designed to cause message collisions, resulting in the victim's error counters incrementing. Lastly, the simulation must track the victim's transition from an error-active state to an error-passive state, and finally to the bus-off state, effectively isolating the victim ECU from the network. This simulation highlights the risks and feasibility of such attacks in real-world scenarios and underscores the importance of enhancing security in vehicular communication systems.

System Setup

The simulation, based on the source code linked above, employs a custom Python framework to emulate the CAN bus network, the ECUs, and their interactions. This ad-hoc solution ensures flexibility in modelling the intricate error-handling dynamics of the CAN protocol, while also allowing for precise control over the simulation parameters, such as bus framerate and frame periodicity. It follows a detailed description of the key components and their roles in the system.



The **CANBus Class** serves as the backbone of the simulation, providing a shared communication channel for the ECUs to interact. It simulates the core features of a CAN bus, including message arbitration, collision handling, and error management. The class maintains an ongoing list of current transmissions to detect and handle collisions to provide the right access to the bus. Key functions include:

- **send_frame**: Adds a frame to the list of ongoing transmissions on the bus.
- **handle_arbitration**: Resolves conflicts between competing messages based on their priority, which is determined by the CAN protocol's ID-based arbitration rules.
- **resolve_collisions**: Detects and manages message collisions by applying error-handling rules, including incrementing error counters and determining the transmission outcomes.
- **receive_frame**: Retrieves the next frame from the bus and processes it according to the resolution of any collisions.

The **ECU Class** models the behaviour of a generic ECU connected to the CAN bus. Each ECU has attributes for tracking its error state, including a transmit_error_counter (TEC) and flags indicating whether it is in an error-passive or bus-off state. These attributes are updated based on the outcomes of message transmissions and collisions. The ECU transitions between states according to CAN protocol rules: error-passive when the TEC exceeds 127, and bus-off when the TEC exceeds 255. This class provides methods for sending and receiving frames, as well as for incrementing and decrementing error counters based on success or failure in transmitting messages.

The **VictimECU Class** extends the ECU class to emulate the behaviour of a victim ECU, which transmits a mix of periodic and non-periodic messages. The victim ECU sends three types of frames: preceded frames, which are transmitted before periodic messages to establish a predictable pattern; periodic frames, which are sent at regular intervals; and non-periodic frames, which are transmitted with random IDs and data values. This behaviour creates a traffic pattern that the attacker ECU can analyse and exploit.

The **AttackerECU Class** represents a compromised ECU with the capability to monitor and manipulate the CAN bus traffic. It observes and analyses the traffic to identify patterns, specifically the periodic messages sent by the victim ECU. In this simple implementation we focused on the preceded ID approach, finding a consistent pattern formed by a periodic frame preceded, most of the time, by the same frame. This is fundamental to execute the attack at the right time in a way to trigger the CAN bus error-handling mechanism and bring the victim to bus-off state. If a consistent pattern is identified, the attacker synchronizes its malicious transmissions with the victim’s periodic messages. By crafting frames with a dominant bit mismatch, the attacker deliberately triggers errors in the victim’s transmissions, causing the victim’s TEC to increment. The attacker continues this process, exploiting the CAN protocol's error-handling mechanism to escalate the victim ECU's error state until it reaches the bus-off condition. If a consistent pattern is not identified, the attack is aborted since there is no way for the attacker to inject the malicious frame at the right time, do not respecting so one of the main requirements for a successful bus-off attack.

The **simulation** operates in two distinct phases. In the Pattern Analysis Phase, the victim ECU transmits its frames as per its predefined behaviour, while the attacker ECU monitors the traffic

to identify periodic messages and their preceding frames. Using this information, the attacker determines the optimal timing and content for its malicious transmissions. In the Attack Phase, the attacker executes the bus-off attack by injecting malicious frames that directly interfere with the victim's periodic transmissions. The CAN bus class handles the resulting collisions and applies the error-handling rules to increment the victim's TEC. Over successive collisions, the victim transitions from error-active to error-passive, and eventually to the bus-off state, isolating it from the network.

The overall simulation flow begins with the victim ECU transmitting its mix of preceded, periodic, and non-periodic messages. The attacker listens to this traffic, identifies patterns, and uses this knowledge to plan its attack. During the attack, the attacker synchronizes its transmissions to coincide with the victim's periodic messages, ensuring that its frames cause deliberate collisions. These collisions are resolved by the CAN bus arbitration logic, which increments the TEC of the victim ECU while allowing the attacker to recover and prepare for subsequent transmissions.

The simulation demonstrates the attacker's ability to exploit vulnerabilities in the CAN protocol to execute a successful bus-off attack. The victim ECU's eventual isolation from the network confirms the attack's feasibility and effectiveness, serving as a reminder of the need for improved security measures in vehicular networks. This custom simulation framework provides a tool for studying such attacks and exploring potential countermeasures.

Experiments

Using our simulation of the CAN bus we were able to replicate the majority of the results that the authors proposed in the paper when talking about the evaluation of the attack on the CAN bus prototype.

In order to keep track of variables that were changing over time, we decided to introduce a new class *setup_logger.py* that was used to log the attacks and retrieve useful data in order to conduct experiments later. Thanks to the logging module provided by standard library of Python we were able to write, for each attack, the information in a logger file that can be found inside the folder *Simulation/attack_logs/attack_log.log*.

```
2024-12-21 09:20:01,206 ECU's name: Attacker's TEC value: 98 Passive error state: False Bus-off state: False
2024-12-21 09:20:01,207 ECU's name: Victim's TEC value: 251 Passive error state: True Bus-off state: False
2024-12-21 09:20:01,207 ECU's name: Victim's TEC value: 250 Passive error state: True Bus-off state: False
2024-12-21 09:20:01,207 ECU's name: Victim's TEC value: 249 Passive error state: True Bus-off state: False
2024-12-21 09:20:01,207 ECU's name: Victim's TEC value: 248 Passive error state: True Bus-off state: False
2024-12-21 09:20:01,208 ECU's name: Victim's TEC value: 256 Passive error state: True Bus-off state: False
2024-12-21 09:20:01,208 ECU's name: Attacker's TEC value: 97 Passive error state: False Bus-off state: False
2024-12-21 09:20:01,208 ECU's name: Victim's TEC value: 255 Passive error state: True Bus-off state: True
2024-12-21 09:20:01,209 End of the 1-th attack
```

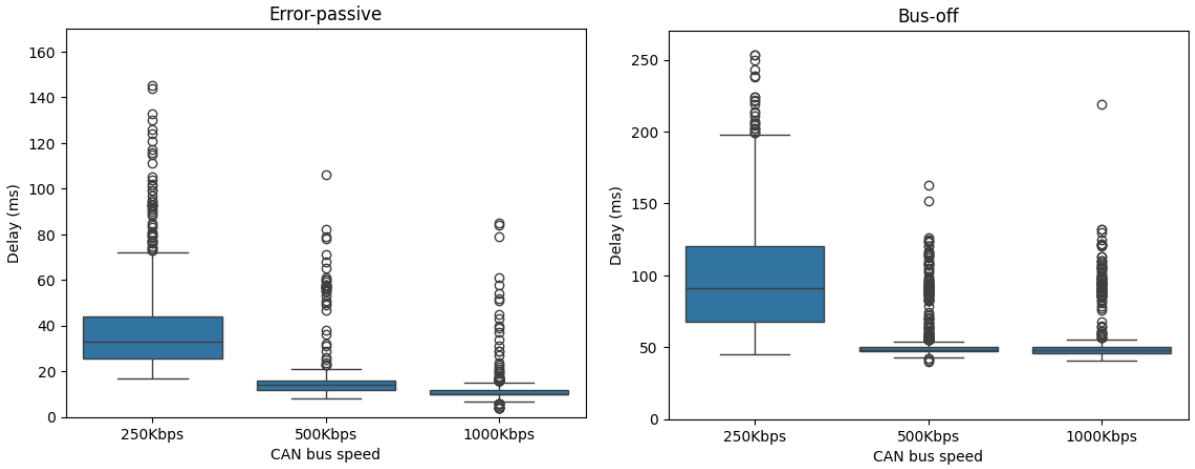
As we can see in the above picture, we considered to log the following information: timestamp of each frame sent, name of the ECU that is sending the frame, TEC value of that ECU, Passive error state and Bus-off state.

As the authors of the paper, we have executed 1000 times the attack varying the bus speeds obtaining the following results. We have analysed the delay in milliseconds (ms) that takes to go to: error-passive state from error-active state and bus-off state from error-passive state. This was done varying the bus speed from 250Kbps, 500Kbps to 1000Kbps as in the reference paper.

The code regarding the experiments and meaningful graphs for the confirming the results can be found in the repository under *Notebook/attack_graphs.ipynb*. Within the *Notebook* folder we can find also logs that was produced while executing the attack 1000 times varying the bus speed, and of one single attack using default parameters.

Analyzing the results of the 250Kbps and 500Kbps cases the attack success rate was 100%, while for the 1000Kbps case we noticed that some attacks were failing. This was because the victim did not go in error-passive state or victim went in error-passive state and did not go in bus-off state. However, on average, the failures were just a few as we can see from the bar chart provided in the notebook. Analyzing the frames of the log of failed attacks when increasing the speed to 1000 Kbps, it is possible that our simple pattern detection algorithm, during the time interval in which the i-th attack occurs, detects the wrong pattern due to many frames being sent, which can mislead the algorithm.

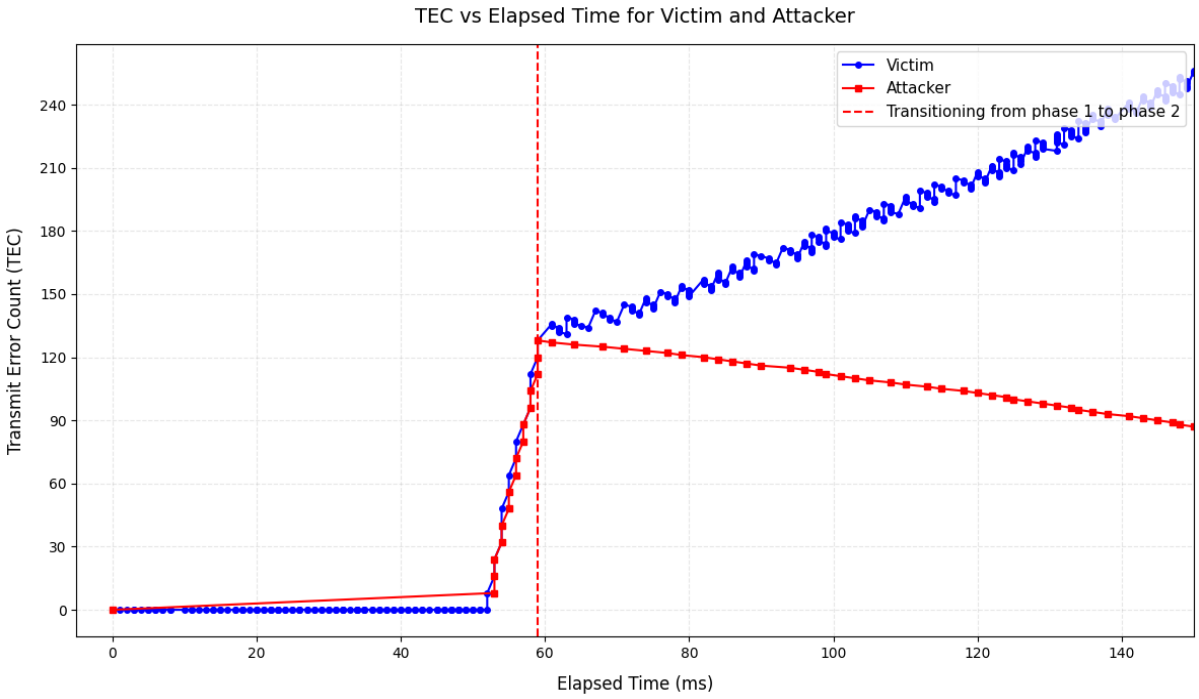
During the experiments we maintained consistent parameters across all trials, varying only the CAN bus speed to analyze the effect of speed on the attack's impact.



The results show that at 250Kbps, the delay exhibits significantly higher variability compared to 500Kbps and 1000Kbps. This suggests that lower bus speeds may increase the system's susceptibility to timing inconsistencies under attack conditions. At higher speeds (500Kbps and 1000Kbps), the delay is more consistent, and transitions to error-passive state and subsequently bus-off state occur more rapidly. This indicates that while higher speeds reduce variability, they also accelerate the system's progression to critical error states.

Results and Discussion

As the results of the analysis of a single attack and the experiments in the notebook show, we can confirm that we were able to approximately replicate the results achieved by the authors of the reference paper when working with a CAN bus prototype. Specifically, the notebook includes the following plots, which illustrate the monotonic increase in the TEC values of both the Victim and the Attacker during Phase 1, as well as the transition from Phase 1 to Phase 2. In Phase 2, the TEC values of the Attacker and the Victim monotonically decrease and increase, respectively, confirming the success of the attack.



We have also confirmed the fact that when we are observing the Phase 1 of the attack the TEC value of the Victim monotonically increase, while In Phase 2 of the attack, we can see a stepwise increase in the TEC value of the victim. This stepwise pattern occurs because the Victim succeeds in its transmission sending recessive error bits (the TEC of the Victim is increased by 8 and then immediately decreased by 1).