# Intro

Cars nowadays are a collection of many computers (or electronic control units - ECUs) interconnected by a network called SBP (Simple Bus Protocol).

The SBP (Simple Bus Protocol) protocol was not designed with security in mind. Numerous attacks can be performed by a rogue component that is connected to a car’s network bus. Such a component can transmit messages that will directly influence other legitimate components: speedometer can show abnormal speeds, and even brakes and steering wheel can be manipulated.

# SBP Protocol Description

* Each ECU transmits messages that are received by all the other units.
* Each unit transmits messages with some frequency all the time
* Messages can be sent at the same time.
* Each message has an ECU ID (uint32), value1 and value2 (each uint32)
* Values are always sent, but not necessarily used (data may be ignored)
* All values have a range of validity

Our car network has the following ECUs connected to it :

|  |  |  |  |
| --- | --- | --- | --- |
| **Unit Name** | **ID (32bit)** | **Value1 (32bit)** | **Value2 (32bit)** |
| **Speedometer** | 0x100 | 0 | CURRENT\_SPEED\_VALUE (0-300) |
| **Pedals** | 0x200 | ACCELERATION\_VALUE (0-100) | BRAKE\_VALUE (0-100) |
| **ABS** | 0x400 | 0 | INACTIVE / ACTIVE (0-1) |
| **TIRE\_PRESSURE** | 0x800 | 0 | PRESSURE\_PERCENTAGE (0-100) |

# The Task

We'll develop a component that will be connected to the car’s SBP network, monitor all the messages transmitted over it and report abnormal behavior.

The component will receive a textual file with messages broadcast on the network and will detect various abnormal behaviors.

* Each message will start on a new line (maximum line length is 100 chars)
* Each message consists of:
  + A message id which is line number (starts from 1)
  + A timestamp in millisec (always starts from 0)
  + An ECU id
  + Value1
  + Value2
* The above fields in each message are separated by space
* All of the above fields are uint32 For example :

1 0000000001 0x100 0 80

2 0000000323 0x200 0 0

3 0000050001 0x100 0 82

4 0000100323 0x200 53 0

5 0000100323 0x400 0 0

Our mission is to detect anomalies which will tell us that either there is a rogue component pretending to be someone else or one of the existing components was hijacked by a hacker. Below are the descriptions of such anomalies, each will require us to implement an interface. The interfaces for all 3 tasks are similar and will require you to report the **first** 1000 anomalous messages (or less if there were less). Ids should be reported in increasing order.

1. **Timing Anomaly -** Detect abnormal frequency. Each ECU transmits messages constantly with frequency that **is not higher** than the one defined below. There can be messages that are missing, but there can not be message that is sent with timestamp less than the frequency defined below. For example, there will be **no more than** one speedometer message every 50 millisec
   * **Speedometer**: transmission frequency - one message every 50 millisec.
   * **Pedals**: transmission frequency - one message every 5 millisec.
   * **ABS**: transmission frequency - one message every 10 millisec.

* **Tire Pressure:** transmission frequency - one message every 100 millisec.

## int detect\_timing\_anomalies(const char\* file\_path, unsigned int \*anomalies\_ids)

1. **Behavioral Anomaly -** Detect abnormal values. Physical limitations impose that:
   * Make sure all the values are within the given range
   * The minimum time a **pedal** can be pressed (value>0) is 10 milliseconds

* The **gas** and **brake** pedal cannot be pressed simultaneously
* The car’s **speed** may not change faster than 5 kmh within 50 milliseconds

i. An exception to this is a car crash, in which the speed will change to 0. After that, it should stay 0 for the rest of the recording. Any positive speed after there was a crash should be treated as an anomaly.

## int detect\_behavioral\_anomalies(const char\* file\_path, unsigned int \*anomalies\_ids)

1. **Correlation Anomaly -** Detect abnormal correlations between various units:
   * When **acceleration** is active ( > 0), speed **should not** be decreasing. In this case only wrong speed is considered an anomaly
   * When **brakes** are active ( > 0), speed **should not** be increasing. In this case only wrong speed is considered an anomaly
   * When **brake** pedal is pressed hard (80+), ABS will activate (value2 is 1). It will remain active as long as the brake pedal is pressed hard. 0 value for ABS in this case is considered an anomaly

* When **tire pressure** is below 30, speed can’t be above 50 kmh. Speed above 50 kmh reported after tire was below 30 is considered an anomaly. Tire pressure below 30, when speed was above 50 is also considered an anomaly.
* Tire **pressure** can’t increase while the car is moving. Abnormal tire pressure is considered an anomaly.

## int detect\_correlation\_anomalies(const char\* file\_path, unsigned int \*anomalies\_ids)

Additional instructions:

* There are some test cases attached in example\_inputs folder, the name of the test case implied the anomalies id it contains.
* Each subtask is completely independent and does not depend on other tasks. Assume that each interface is supposed to deal only with anomaly it is supposed to detect and the log file that will be provided to it can only have anomalies of that type For example **detect\_behavioral\_anomaly** will never be invoked on log file with **timing anomalies** or **correlation anomalies**
* If some message is anomalous, in addition to reporting it, you still need to take it into account when looking at anomalies in other messages. For example if brake and gas were pressed simultaneously, and it happened for less than 10 millisec, you will report 2 anomalies, one in pedal message that has both gas and break and the other one in the next pedal message that has 0 in either gas or break (or both) within less than 10 millisec.
* You can assume that there were no events before the start of the log
* You can assume that the initial value of each ECU is 0, **except for Tire pressure** that has the initial value of 100.
* The value of each ECU remains the same till next message from that ECU is received For example if there are the following messages in the log :

0000000300 0x200 0 0

0000000305 0x200 0 70

0000000310 0x200 0 55

It means that initially brake had the value 0**,** it remained the same till 300, it was 0 till 305, at 305 it was activated with value 70, it stayed the same till 310, then at 310 it was activated with value 55 and will stay with the same value till next message from brakes