Eclipse KUKSA.val for SCR Anti-Tampering Monitoring in Heavy Vehicles

Junhyung Ki
Dortmund University of Applied Sciences and Arts
`junhyung.ki001@stud.fh-dortmund.de`

Sebastian Schildt
Robert Bosch GmbH Corporate Research
`sebastian.schildt@de.bosch.com`

Andreas Hastall
Robert Bosch GmbH Powertrain Solutions
`andreas.hastall@de.bosch.com`

Sven Erik Jeroschewski
Bosch.IO GmbH Expert Squad - Open Source
`svenerik.jeroschewski@bosch.io`

Robert Höttger
Materna Information & Communications SE BL Digital Transformation
`robert.hoettger@materna.de`

***Abstract*—Modern internal combustion engines have several advanced exhaust treatment systems to meet emission standards and legislation. In case of Selective Catalytic Reduction (SCR) for diesel engines, a catalyst (“AdBlue”) is used as consumable. This incurs costs for the operator of diesel vehicles and provides an incentive to unlawfully circumvent and shut down those systems.**

This case study presents how the Eclipse KUKSA stack has been used to realize an anti-tampering system for commercial heavy-duty trucks exhaust systems. We show, how the in-vehicle KUKSA.val software and the KUKSA.cloud components can be used to collect relevant data from a real heavy-duty truck and send them to the cloud for further analysis.

I. INTRODUCTION

Modern vehicles with internal combustion engines are equipped with exhaust treatment systems that drastically reduce the emission of harmful exhaust gases. Modern exhaust treatment systems for diesel engines include an SCR (Selective Catalytic Reduction) system to reduce emission of nitrogen oxides (NO_x). An SCR converts NO_x into harmless nitrogen (N_2) and water (H_20) with the help of a catalyst fluid. The catalyst, an urea ($CO(NH_2)_2$) - water solution, is a fluid, also called “Diesel exhaust fluid” (DEF) [1] or “AdBlue”

Just like the fuel itself, the DEF is a consumable. Costs for an operator can reach up to 1500 USD/year for a commercially operated heavy truck. This provides the incentive to maliciously interfere with the correct operation of the exhaust treatment system. There are companies that offer facilities and services to disable these systems. Common tampering methods are small hardware modules connected to internal busses or diagnostics interfaces of a vehicle injecting faulty data regarding the level of the catalyst or measured NO_x values, combined with disabling components of the exhaust treatment. For common engines, tampering hardware (see Figure 1 for an example) can be obtained for a one-time investment of less than 30 USD. A vehicle modified in such a way can still operate with its full performance but will emit harmful substances significantly above the legal limit. Such manipulation is difficult to detect, without randomly inspecting trucks on the road, which is cost- and time consuming and does not scale.

DIAS[1], a joint European research and development project, has the goal to help prevent or uncover these manipulations. The DIAS approach to detect tampering is two-fold: Inside a vehicle, data from various sensors is gathered and a validation of the gathered data can be performed to detect values inconsistent with current operating conditions. As computational power in a vehicle is limited and it is to be expected that tampering methods get more intricate (this has already happened in the past), the gathered data is transmitted to a cloud backend, where a more complex analysis over longer time frames is possible.

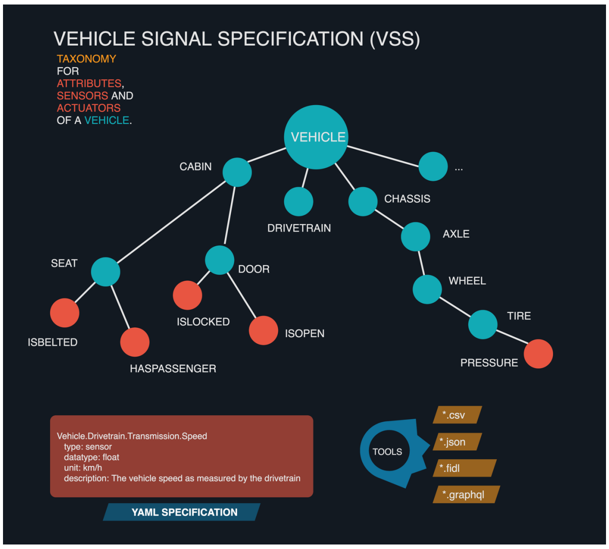
Drive Your Truck Without Diesel Exhaust Fluid
adBlue OBD2 ™
for FORD Trucks
adBlue/DEF & NOx Emulator via OBD2
PLUG & DRIVE READY DEVICE

Fig. 1: Typical AdBlue Emulator

Getting data from a vehicle in a safe and secure manner is a challenging and recurring task when developing applications for connected vehicles. Not only is this task very complex, most of the time the solution is not portable due to the heterogeneity and specifics of the underlying system architecture. The KUKSA.val project aims to ease this task by abstracting

---

[1] https://www.dias-project.com/



```
Vehicle.Drivetrain.Transmission.Speed
  type: sensor
  datatype: float
  unit: km/h
  description: The vehicle speed as measured by the drivetrain
```

Fig. 2: Genivi VSS structure

the underlying systems through the provision of a server with which other applications in the vehicle can interact based on a standardized data model and API. By doing so applications can collect vehicle data in a safe, secure and, most importantly, portable manner.

In this case study we outline the system architecture and data model for an integrated system to detect exhaust treatment tampering and present a prototype that has been created using components from Eclipse KUKSA which is an open source software stack for building connected vehicle ecosystems. We present how to access data in a heavy-duty truck and introduce the necessary extensions enable the KUKSA data feeder component to work with heavy-duty vehicles. Finally, a working prototype of the system has been tested in a real heavy-duty truck.

II. BUILDING BLOCKS

In our system we use various existing technologies and standards, which we will explain in the following.

A. *Genivi VSS*

The Genivi Vehicle Signal Specification (VSS) [2] introduces a domain taxonomy for vehicle signals. The goal is to create a common understanding of vehicle signals in order to reach a "common language" for vehicle data independent of the protocol or serialization format. VSS can be used as standard in automotive applications to communicate information related to the vehicle, which is semantically well defined. It focuses on vehicle signals, in the sense of classical sensors and actuators usually connected to the "deeply-embedded" ECUs as well as data which is more commonly associated with the infotainment systems. A simplified structure of the VSS model is shown in Picture 2.

A VSS tree consists of three basic types of nodes: *Branches* describe the hierarchy of signals (e.g. `Vehicle.Cabin`), *sensors* describe values that are expected to change during the operation of a vehicle (e.g. `Vehicle/Speed`), and *attributes* that are expected to be static over the lifetime of a vehicle (e.g. `Vehicle/VehicleIdentification/VIN`).

B. *W3C VISS*

While VSS describes the structure of signals in a vehicle, the W3C Vehicle Information Service Specification (VISS) [3] defines a websocket-based protocol to access such signals using either a query response pattern or publish/subscribe mechanism. Currently, the second iteration of VISS, VISS2 is under development[2]. Being a network-based API VISS is a good fit for modern Vehicle Computer Architectures, where different safety and security zones are separated by hypervisors or container technologies. Instead of linking to software components, they can be accessed through the network by other (micro-)services running in other containers, hypervisor domains or computers. Basic encryption, authentication and integrity can be provided by using standard TLS mechanisms.

C. *Eclipse KUKSA*

Eclipse KUKSA provides building blocks for the creation of ecosystems around applications in connected vehicles. On a high-level Eclipse KUKSA differentiates between the in-vehicle side and the cloud backend. The in-vehicle platform allows and simplifies the execution of applications in a vehicle. With the cloud backend it is possible to handle the data originating from the in-vehicle applications. In addition, KUKSA.cloud can manage the distribution and roll-out of applications to the vehicle.

D. *Eclipse KUKSA.val*

Eclipse KUKSA.val[3] is an in-vehicle component from the Eclipse KUKSA stack. KUKSA.val is a server that manages VSS data and provides them via the VISS protocol to other applications running in the vehicle in a safe and secure manner. Written in C++, KUKSA.val offers a small footprint making it suitable running in a vehicle computer. KUKSA.val implements Version 1 of the VISS protocol with some extensions, most notably a security mechanism based on JSON Web tokens [4] providing fine-grained access control to each element in the VSS tree. Additionally, it supports dynamic extension/modification of the VSS tree during runtime and provides a Python library wrapping the VISS websocket protocol to simplify application development. KUKSA.val is optimized to run in a light-weight container environment.

E. *Eclipse KUKSA.cloud*

The cloud backend of the Eclipse KUKSA ecosystem[4] is a composition of multiple open source projects and KUKSA specific components. Many of the adopted technologies are coming from the community around the Eclipse IoT working group. The current version of the KUKSA.cloud is tailored to run in a Kubernetes environment and can be deployed with a

---

[2] https://github.com/w3c/automotive

[3] https://github.com/eclipse/kuksa.val

[4] https://github.com/eclipse/kuksa.cloud

KUKSA.cloud specific Helm Chart. Besides the management of applications on a vehicle, the KUKSA.cloud allows the ingestion of data coming from the vehicles. In this case study we are mainly dealing with the data aspect where the data ingestion is covered by running Eclipse Hono [5] within the KUKSA.cloud.

Eclipse Hono is a communication hub that provides interfaces and endpoints for connecting a large numbers of IoT devices to a backend and enables interaction with them in a uniform way regardless of the device communication protocol. In that regard, Eclipse Hono has a "southbound" API designated to be used by the IoT devices like in our case heavy-duty vehicles and a "northbound" API which allows other applications in a cloud backend to consume the data coming from the devices. Similarly, it is also possible to send data through the northbound API from the cloud backend to devices connected at the "southbound" API. To support a wide range of devices and implementations, Eclipse Hono offers the "southbound" API for a number of protocols like HTTP, MQTT or CoAP through different protocol adapters which are implemented as individual micro-services. These protocol adapters internally convert the messages to the AMQP 1.0 protocol which is currently used for Hono's "northbound" API. Since Eclipse Hono is designed to be highly scalable, it is a feasible option for ingesting data from a large fleet of vehicles.

Furthermore, we are using a Hono-InfluxDB connector [6] that is maintained within the KUKSA.cloud project . This connector listens at the "northbound" API of Eclipse Hono for new data and then writes it into the time-series database, InfluxDB [7]. Storing the data into a database makes it easily accessible for further processing or visualization like in our case using a Grafana[8] dashboard.

F. SAE J1939

While VSS, VISS and KUKSA.val offer useful abstractions dealing with vehicle data and enable rapid function development, the majority of data in a vehicle originates from deeply embedded ECUs connected to low bandwidth busses such as CAN [5]. Standard CAN is a simple two-wire communication protocol where messages are identified by 11 or 29 bit identifiers including up to 8 bytes of payload.

In addition to CAN, SAE J1939 [6] standard is a higher level protocol that uses the CAN Bus technology as a physical layer. It is used for communication and diagnostics among vehicle components. Originating in the car and heavy-duty truck industry in the United States, it is now widely deployed in heavy-duty vehicles around the world. In addition to the standard CAN Bus capabilities, SAE J1939 supports node addresses, and it can deliver data frames longer than 8 bytes (up to 1785 bytes). Signals are not identified by the raw CAN ID, but by a Parameter Group Number (PGN), that is encoded in the 29 bit CAN identifier, where data belonging to a PGN can span multiple CAN messages.

III. ANTI TAMPERING SYSTEM

As lined out in the introduction, a resilient anti-tampering system relies on in-vehicle components as well as a powerful cloud backend. In the vehicle data needs to be collected and transmitted. As upcoming vehicle platforms include powerful computing units, data can be processed on-board to detect many naive forms of tampering. However, compared to the cloud, a vehicle's computational resources are still limited. Sending gathered data to a cloud backend gives the chance to run more complex algorithms. For example more precise engine models and longer time frames can be taken into account. Also, even with modern vehicle computers it is to be expected that cloud services can be updated faster.

A. Architecture

Figure 3 shows the overall system architecture and components. The prototype is built around a Raspberry Pi 4 SBC running Linux, allowing a desk setup as well as connecting to a vehicle. Additionally, a Pi is roughly comparable to upcoming vehicle computers in terms of computing power and memory. Data is received using the standard Linux socket CAN interface. This way either simulated CAN traces can be played or the Pi can be connected to an existing CAN network. Raw CAN/J1939 data is processed by the *DBCFeeder* (see Section III-C) and converted to valid VSS signals that are fed into the KUKSA.val server.

The *cloudfeeder* component connects to the KUKSA.val server and collects the required data in VSS format via VISS. It will then upload the collected data to the cloud. Optionally, (pre)processing on the vehicle is possible.

B. VSS Model

Not all the signals required for monitoring the exhaust system are part of the VSS standard catalog. However, the standard tree provided by VSS can be extended with custom signals. Figure 4 shows all signals collected for our exhaust treatment monitoring. While algorithms for anti-tampering detection probably need to be parameterized differently depending on each specific engine type, the input data required from a vehicle will be similar. By mapping data to a common VSS model, the software module for accessing data and transmitting it to a cloud backend can be reused across different vehicles.

C. Reading J1939 data

As mentioned in Section II-F, relevant data from heavy-duty vehicles is available via the J1939 protocol based on CAN. To read data from a real truck we equipped the Pi with a dual channel CAN shield[9]. As most vehicles have several CAN buses and not all data is available on all of them, a dual channel shield gives the opportunity to easily tap two busses at once.

---

[5] https://www.eclipse.org/hono/

[6] https://github.com/eclipse/kuksa.cloud/tree/master/utils/hono-influxdb-connector

[7] https://github.com/influxdata/influxdb

[8] https://grafana.com/oss/grafana/

[9] https://wiki.seeedstudio.com/2-Channel-CAN-BUS-FD-Shield-for-Raspberry-Pi/

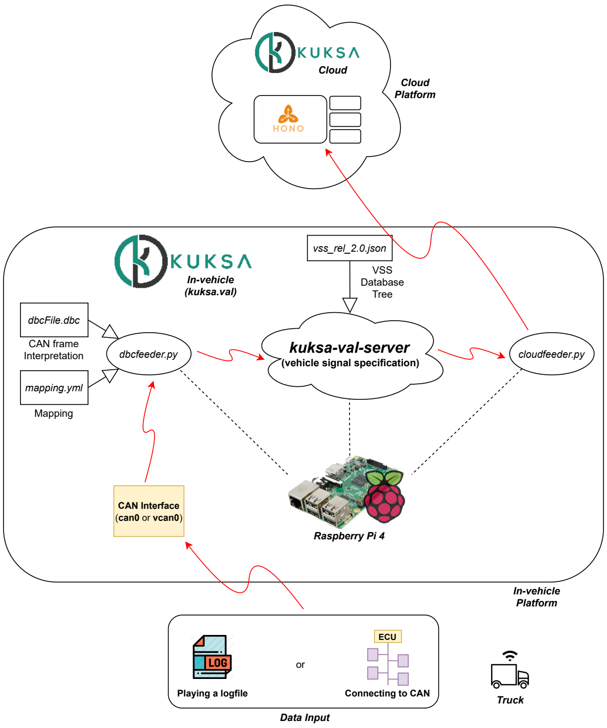


Fig. 3: Exhaust Anti-Tampering System Architecture

KUKSA.val already offered a python-based DBCFeeder component to read CAN data and map it to the VSS tree. A DBC file is an ASCII file describing which CAN frames contains which signals and what conversions are required. This is important, as most CAN data is not directly encoded in SI units, but instead the range and resolution is space-optimized for a specific use case, so often an offset and scale factors need to be applied. However, we discovered the DBCFeeder was only able to deal with raw CAN frames, and was not J1939 aware.

For this case study we extended the KUKSA DBCFeeder with J1939 support. As the DBCFeeder included in KUKSA.val is Python-based, we leveraged support of an existing Python J1939 implementation[10]. When dealing with a J1939 system and an associated DBC file describing J1939 PGN units, the DBCFeeder can be started with the "-j1939" option. This will switch the Raw CAN reader normally used by the DBCFeeder with the J1939 stack reassembling PGN data from the raw CAN frames, before decoding and mapping the data to VSS. As multiple DBCFeeder instances can run in parallel, you can process raw CAN frames or J1939 PGN data at the same time (see Figure 5). The initial J1939 support developed has also been merged upstream, so it is available to all KUKSA.val users.

D. Processing and Transmitting

CloudFeeder is a module that retrieves the observed signals' values from the in-vehicle KUKSA.val-server, pre-processes the retrieved data using a custom pre-processor script, and finally transmits the result data to the cloud via MQTT. The pre-processor script can be changed depending on the kind of processing that is desired on-board the vehicle.

Figure 6 shows the sequence of how the *CloudFeeder* works. In our setup the *CloudFeeder* is set up to transmit data to an Eclipse Hono instance running in the cloud.

E. Cloud setup

The *CloudFeeder* uses MQTT to connect to an Eclipse Hono MQTT protocol adapter. We are using the KUKSA.cloud InfluxDB Connector, which receives data from Hono and puts it into the time-series database InfluxDB.

A custom diagnostic routine [11] can access the data from the InfluxDB and perform analysis on historic data to detect anomalies that can not be detected in a vehicle. The cloud analytics can be updated more frequently to keep up with novel tampering methods, and it can perform more compute intensive analytics, such as incorporating detailed models of a specific internal combustion engine. Having the data of a larger number of vehicles at hand also offers the potential to apply more sophisticated algorithms for automated anomaly detection.

The raw and processed data stored in InfluxDB can also be monitored in custom Grafana Dashboards (see Figure 7c).

IV. REAL-WORLD TEST

To validate our setup we tested it in a real truck. The test vehicle used by the DIAS project is a Ford Otosan F-MAX heavy duty truck (Figure 7a. This truck is modified, so that the in-vehicle CAN busses and measurement equipment is available for easy access in the cabin (Figure 7b). It is important to note, the truck's ECUs do not need to be modified. The only component required to enable this use case is the Raspberry connected to the vehicle's busses providing a runtime for the in-vehicle part of the software stack and Internet access. In a series production vehicle, it is expected, that the software running on the Pi will be running on a Vehicle Computer inside the truck alongside other services. Figure 7c shows a Grafana dashboard that is updated live during the test drive.

During the test drives in the Stuttgart region the system performs as expected. The major challenge is dealing with intermittent Internet connectivity. While our simple prototype still has some potential for optimization in this regard, it is not a blocker for the DIAS use case: Anti-tempering monitoring does not require real-time data. Real-time analysis is done directly on the vehicle, and any errors are logged similar to other error conditions. In cases of no connectivity, data can be cached in the vehicle and transmitted once connectivity is available again.

The availability of connectivity hardware can be expected for all vehicles in the near future. In fact, many heavy-duty trucks already contain some form of connectivity for fleet

---

[10] https://pypi.org/project/j1939/

[11] https://github.com/junh-ki/dias_kuksa/

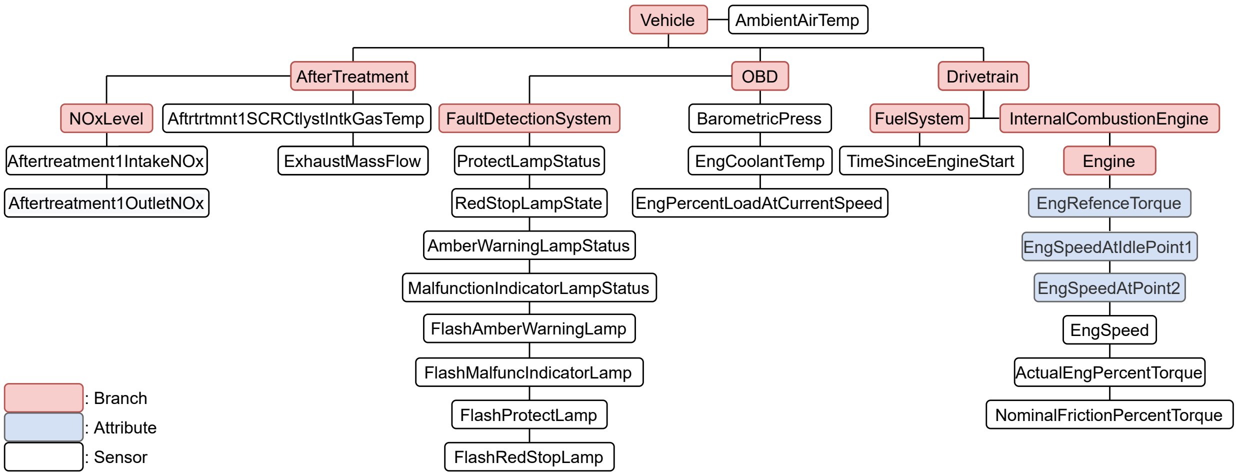


Fig. 4: VSS signals used for exhaust treatment monitoring

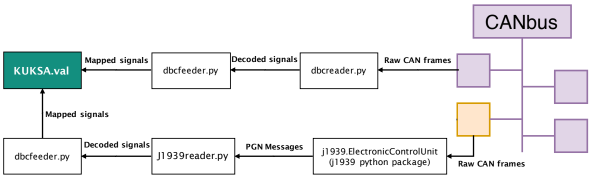


Fig. 5: Two dbcfeeders feeding a single KUKSA.val instance

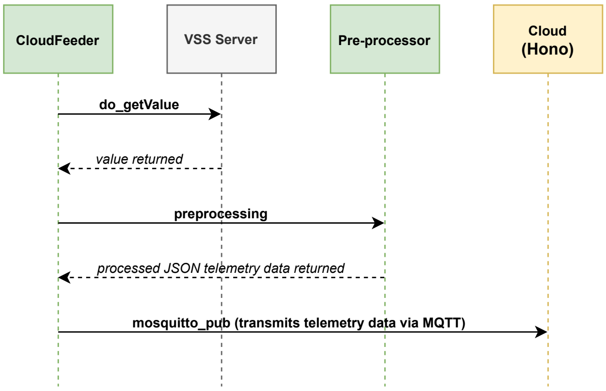

CloudFeeder
VSS Server
Pre-processor
Cloud
(Hono)
do_getValue
value returned
preprocessing
processed JSON telemetry data returned
mosquitto_pub (transmits telemetry data via MQTT)

Fig. 6: Cloudfeeder sequence diagram

management purposes. Additionally, existing or upcoming legislation might require to provide connectivity hardware to a vehicle: In the EU, the eCall regulation [7] already requires cellular connectivity in every passenger vehicle type-approved after March 2018, China already requires telemetry for all EVs [8] and a legislative initiative by the German ministry of transport aims to require that all vehicles with autonomy functions have to be connected "all the time" [9].

V. CONCLUSION & OUTLOOK

We have introduced the problem of exhaust treatment system tampering. Potential cost savings incentivize actors to unlawfully interfere with the correct function of heavy vehicle's exhaust treatment system. The DIAS research project tries to tackle this problem using a combination of on-board and off-board diagnostic of vehicle data. In this case study, we have presented a system for Exhaust System Anti-Tampering monitoring using open source components from the Eclipse KUKSA ecosystem.

We were able to collect the relevant data from heavy-duty trucks, processing them on board the vehicle and transmitting them to the cloud for further analysis. The system has been tested extensively with simulated CAN traces and inside a real heavy-duty truck. To enable this we extended KUKSA.val with J1939 support and provided it upstream.

In the future, the system can be extended with more robust caching during times of no connectivity. Additionally, the data received from CAN busses can be authenticated. While most communication in contemporary vehicle is unencrypted and unauthenticated, there are upcoming standards providing security on the level of automotive field busses such as CAN. One example is AutoSar SecOC [10], that can provide authentication of individual CAN messages. On vehicles supporting such standards, they are a good first line of defense.

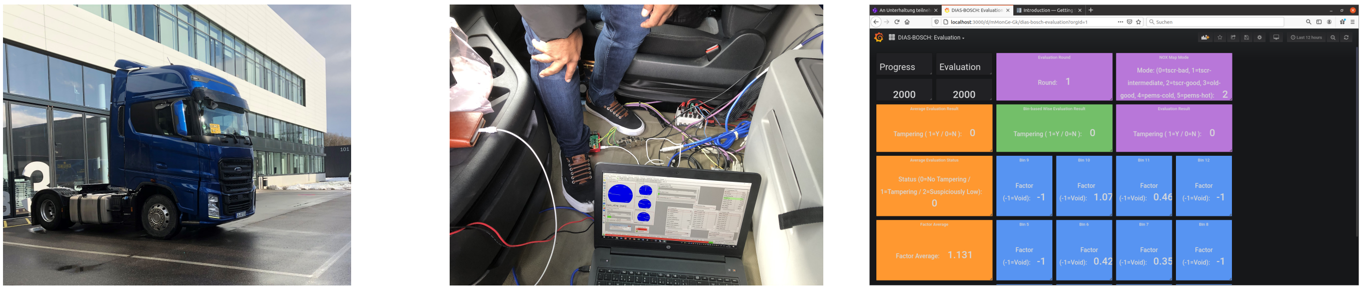
This example showcases the applicability of open source components in the automotive domain. While the technology behind exhaust treatment systems, and advanced tampering detection mechanisms are highly proprietary in nature, it is still possible to leverage the power of open software and standards. By using the open VSS standard and existing open source components on modern vehicle computers, applications for gathering, processing and transmitting data to a cloud backend can be realized in a fast and cost-effective manner.

ACKNOWLEDGMENTS

This paper is supported by European Union's Horizon 2020 research and innovation programme under grant agreement No 814951, project DIAS (https://dias-project.com/).

REFERENCES

[1] "ISO 22241-1:2019- Diesel engines — NOx reduction agent AUS 32 — Part 1: Quality requirements," International Organization for Standardization, Standard, 2019.



Progress: 2000
Evaluation: 2000
Evaluation Round — Round: 1
NOX Map Mode — Mode: (0=tscr-bad, 1=tscr-intermediate, 2=tscr-good, 3=old-good, 4=pems-cold, 5=pems-hot): 2
Average Evaluation Result — Tampering (1=Y / 0=N): 0
Bin-based Wise Evaluation Result — Tampering (1=Y / 0=N): 0
Evaluation Result — Tampering (1=Y / 0=N): 0
Average Evaluation Status — Status (0=No Tampering / 1=Tampering / 2=Suspiciously Low): 0
Bin 9 — Factor (-1=Void): -1
Bin 10 — Factor (-1=Void): 1.07
Bin 11 — Factor (-1=Void): 0.46
Bin 12 — Factor (-1=Void): -1
Factor Average — Factor Average: 1.131
Bin 5 — Factor (-1=Void): -1
Bin 6 — Factor (-1=Void): 0.42
Bin 7 — Factor (-1=Void): 0.35
Bin 8 — Factor (-1=Void): -1

(a) F-MAX test vehicle (b) Setup in cabin (c) Dashboard Overview

Fig. 7: Testing System in a Heavy-Duty Truck

[2] *Vehicle Signal Specification*, GENIVI Alliance Std., 2020. [Online]. Available: https://genivi.github.io/vehicle_signal_specification/

[3] P. Kinney, A. Crofts, W. Lee, and K. Gavigan, “Vehicle information service specification,” Candidate Recommendation, Feb. 2018, https://www.w3.org/TR/2018/CR-vehicle-information-service-20180213/.

[4] M. Jones, J. Bradley, and N. Sakimura, “Json web token (jwt),” Internet Requests for Comments, RFC 7519, May 2015, http://www.rfc-editor.org/rfc/rfc7519.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7519.txt

[5] “ISO 11898-1:2015- Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling,” International Organization for Standardization, Standard, 2015.

[6] *J1939 Standards Family*, Society of Automotive Engineers SAE Std., 2013. [Online]. Available: https://www.sae.org/standardsdev/groundvehicle/j1939a.htm

[7] “REGULATION (EU) 2015/758 concerning type-approval requirements for the deployment of the eCall in-vehicle system based on the 112 service and amending Directive 2007/46/EC,” European parliament and council, Tech. Rep., 2015.

[8] B. Martens and B. Zhao, “JRC Digital Economy Working Paper - Data access and regime competitionA case study of car data sharing in China,” European Commision, Tech. Rep., Aug. 2020.

[9] German Ministry of Transport, “Entwurf eines Gesetzes zur AÄnderung des Straßenverkehrsgesetzes und des Pflichtver- sicherungsgesetzes – Gesetz zum autonomen Fahren,” Feb. 2021. [Online]. Available: https://www.bmvi.de/SharedDocs/DE/Anlage/Gesetze/Gesetze-19/gesetz-aenderung-strassenverkehrsgesetz-pflichtversicherungsgesetz-autonomes-fahren.pdf

[10] *Specification of Secure Onboard Communication*, AUTOSAR Std., 2017. [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SecureOnboardCommunication.pdf