VITROBENCH: Manipulating In-vehicle Networks and COTS ECUs on Your Bench

A Comprehensive Test Platform for Automotive Cybersecurity Research

Anthony Yeo Kee Teck, Matheus E. Garbelini, Sudipta Chattopadhyay, Jianying Zhou

***Abstract*—With the increasing connectivity employed in automotive systems, remote cyber attacks have now become a possibility and concrete threat. Prior works on automotive cyber security solutions have primarily focused evaluation either on real cars or via emulations of electronic control units (ECUs). Evaluation on real cars offer limited flexibility in manoeuvring the packets communicated through in-vehicle network (IVN). Meanwhile, emulations of ECUs rely on assumptions that may not correspond to the exact features in an IVN.**

In this paper, we present VITROBENCH, a comprehensive test platform involving commercial off-the-shelf (COTS) ECUs that allows arbitrary packet control over IVN. In contrast to existing automotive testbed, an appealing feature of VITROBENCH is that it allows replication of driving use cases and scenarios directly on the testbed involving COTS ECUs. This, in turn, allows us to design and evaluate concrete attacks that are directly related to a driving scenario. We present the design of VITROBENCH that allows us to sniff, inject packets to and isolate targeted ECU via bridging. The isolation of ECUs also offers fuzzing the respective ECUs. We evaluate the capability of VITROBENCH via launching concrete attacks and demonstrating the impact of such attacks. We discuss the careful design choices involved in VITROBENCH that inspire automotive cybersecurity research in future.

1. INTRODUCTION

Due to the massive progress in increasing the connectivity of automotive systems, manufacturers of cars can no longer consider the deployed systems to be isolated. Indeed, during the last few decades, the cybersecurity concerns for automotive systems have increased dramatically. Physical and remote attacks on cars are now concrete threats, as exemplified by existing studies [30] and cyber attacks [8]. Recent survey articles [1] have also highlighted the importance of ensuring the security for in-vehicle networks (IVN), among other security concerns for automotive systems. For example, by compromising one or more ECUs in a car, an attacker may freely manipulate the messages communicated through IVNs. This may lead to malicious message being sent to critical car components such as engine and brake. Such a phenomenon may severely impact the normal functions in a car, resulting in serious consequences. Testing of IVNs, to validate possible security concerns, is therefore critically important.

In this paper, we present the design and evaluation of VITROBENCH, a comprehensive test platform involving COTS ECUs to facilitate cybersecurity research for automotive systems, specifically, for in-vehicle networks. Figure 1 contextualizes the contribution of our test platform. Broadly, the existing test platforms can be categorized into three. *Car-based testing* (see Figure 1(a)) leverages a real car for evaluation [31]. As

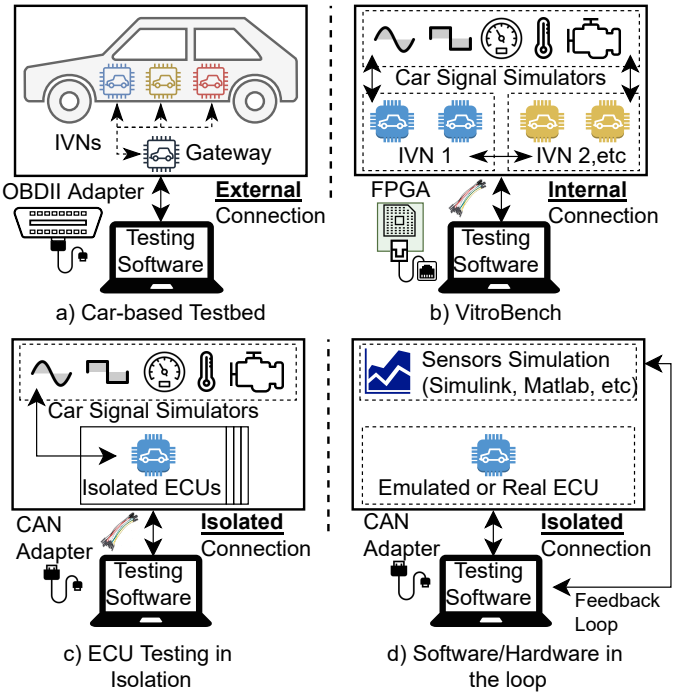

Fig. 1: VITROBENCH vis-á-vis existing test platforms

shown in Figure 1(a), the tests are launched via external OBD-II connector and all communication pass through the gateway ECU. Due to the reliance on external connections, such evaluations provide limited control over the communication traffic in the IVN. Additionally, it is not always cost-effective to use real cars for evaluation and certain cyber attacks might be dangerous to perform on actual cars. Meanwhile, *testing ECUs in isolation* (see Figure 1(c)) only hooks a few ECUs via bare wires [18] but ignores message exchanges in an IVN that can be attributed to realistic driving/stationary environment. This platform is appropriate only to test the specific ECUs and such platforms are incapable to evaluate targeted attacks on IVN that span across multiple ECUs and networks. Finally, Figure 1(d) captures a test platform with *software or hardware in the loop* i.e., a real or simulated ECU. Additionally, a software simulation is used for the rest of the car IVNs via Matlab [2], Simulink [3] and CANoE [7]. While several prior works have considered such test platforms [23, 22] due to its simplicity, these platforms cannot accurately capture realistic IVN messages exchanged between COTS ECUs. This

is because such remaining bus simulation (RBS) via software relies on assumptions that may not correspond to the exact features of the targeted IVN.

The primary objective of VITROBENCH is to facilitate the discovery and investigation of attacks that may impair the *physical functions* (e.g., display of speed, engine functionality) in a car. To this end, capabilities such as sniffing and injection of modified packets are embodied in our test platform. The advantages for such a test platform are multi-fold. Firstly, VITROBENCH allows not only sniffing and replay, but arbitrary injection of original as well as manipulated messages. Indeed, our empirical evaluation shows that message injection is a key capability required for finding concrete attacks on the IVN. Secondly, our test platform allows to create test scenarios based on driving use cases appearing in practice. Thus, original equipment manufacturer (OEM) can test the security of a car based on such practical test scenarios. This, in turn, allows to understand the malicious message flow within an IVN that impacts the critical car functions (e.g., engine functions or display). To the best of our knowledge, VITROBENCH is the first test platform that provides such flexibility using COTS ECUs, yet without access to a real car. Finally, the attacks and security tests discovered by our test platform can be further used for developing effective online mitigation, for example, to detect potential manipulation of messages and raise alarms. In a nutshell, VITROBENCH allows to draw connection between the malicious flow of IVN messages and impairment of car functions. As a result, the designer can pinpoint to an exact sequence of messages to replay, investigate and mitigate attack scenarios *relevant to car functions*.

The abstraction of VITROBENCH design is outlined in Figure 1(b). Specifically, VITROBENCH provides a test platform where we can freely control the communication in IVNs via an FPGA communication board and by hooking COTS ECUs via bare wires. The full control of the IVN communication is facilitated by bridging an ECU, intercepting any message involving the ECU and sending it to a workstation; and then send a possibly modified message back to the IVN. Consequently, leveraging VITROBENCH, designers can perform targeted attacks and advanced fuzz testing on arbitrary COTS ECUs. This is in stark contrast to using real cars where limited control is available via the external connection only. In contrast to the test platforms captured in Figure 1(c)-(d), VITROBENCH fully exposes realistic message frames across multiple CAN networks without requiring an actual car. Such a feature is crucial to design and test arbitrary cross network attacks that would be otherwise damaging on real cars. To the best of our knowledge, VITROBENCH *is the first comprehensive test platform that exposes realistic IVN messages and provides full control of these messages at designer's hand without requiring an actual car, yet involving COTS ECUs.*

VITROBENCH is designed carefully with three components i.e., an automotive testbed, a communication board and a software component. The communication board is used to communicate between the testbed and the software component. It supports multiple IVN protocols e.g., CAN, CAN-FD and LIN. The design of these three components is decoupled in a fashion that the ECUs can be replaced freely in the automotive testbed, keeping the rest of VITROBENCH untouched. Likewise, the communication board and the software component can be modified independently to support, for example, more communication protocols and sophisticated fuzzing algorithms, respectively. An appealing feature of VITROBENCH is that we provide external signals to control and keep the car environment stable during our tests – a feature that is critical for test reproduction, but challenging to ensure using a real car. This makes all our tests reproducible, yet realistic. We hope that VITROBENCH opens the door of automotive cybersecurity research along several directions in future.

After providing an overview and requirement of VITROBENCH (Section 2), we make the following contributions:

1) We detail the implementation and carefully discuss the design choices for VITROBENCH to fully control communication in Control Area Network i.e., CAN (Section 3).
2) We comprehensively evaluate the capability of VITROBENCH. We show the capability of VITROBENCH in terms of sniffing, injecting and intercepting arbitrary messages from IVN (Section 4).
3) We perform multiple case studies leveraging the capabilities of VITROBENCH namely fingerprinting, fuzz testing and targeted attack generation. From fingerprinting, we show that VITROBENCH reliably (with negligible variance) measures inter-frame latency for all messages. Leveraging the information obtained from our fuzz testing, we design five targeted attacks that are directly attributed to real-life scenarios. We show the impact of these attacks on VITROBENCH platform (Section 4).
4) To show the generalizability and extensibility of VITROBENCH, we extend it with CAN-FD ECUs. We show that the capability of our test platform can easily be transferred to a different IVN protocol (Section 4).

We position our VITROBENCH with respect to existing works (Section 5) and discuss our future outlook (Section 6) in terms of inspiring research in automotive cybersecurity. We conclude in Section 7.

2. TEST PLATFORM OVERVIEW

We design VITROBENCH with the *real car ECUs*, *car simulator*, *communication interface*, *host machine or workstation* (for traffic control and analysis) and accompanying *software application*. We select the ECUs for the major operations involved in a car. This includes ECUs for engine, ignition, motion control and instrument cluster. We leverage the car simulator to inject the required signal to the selected ECUs. A multi-channel and multi-protocol communication board interface with the ECUs via a software application running on the workstation.

Broadly put forward, our objective from the VITROBENCH test platform is to investigate the messages of each in-vehicle network and corresponding car response. Specifically, we aim

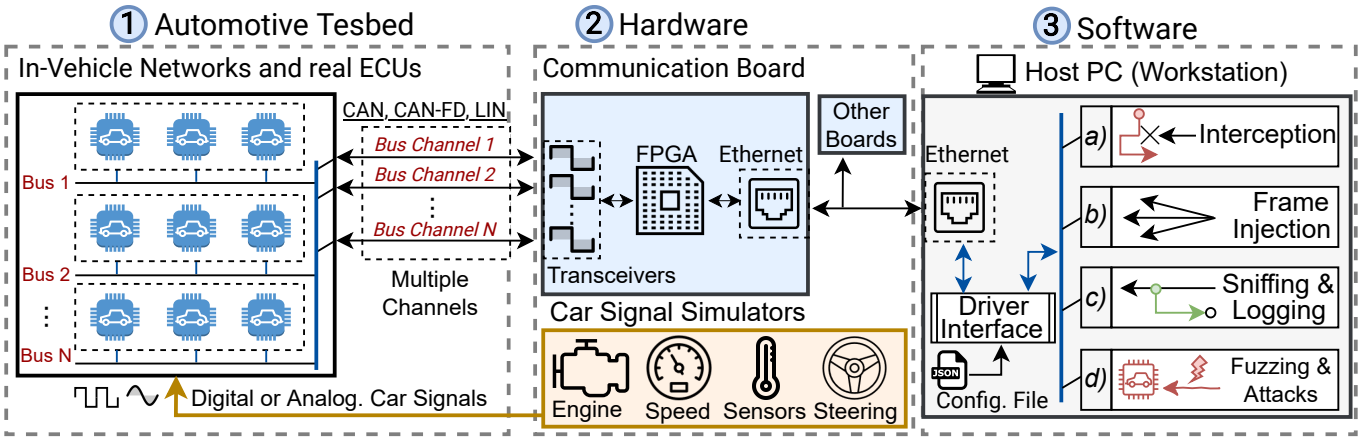

Fig. 2: VITROBENCH Test Platform Architecture

to facilitate the investigation of the following functions via the VITROBENCH test platform:

1) Study the behaviour of the car via observing the messages,
2) Obtain the fingerprint of all the ECUs messages such as source ECUs, frame interval of each message and boot up sequence,
3) Reverse engineering of unknown messages,
4) Search for vulnerabilities via message fuzzing, flooding or spoofing,
5) Conduct cyber-attack and study the impact to the car,
6) Study and research on algorithm for Intrusion Detection and Prevention System.

In the subsequent sections, we detail the design of various components and interfaces within the VITROBENCH.

2.1 VITROBENCH *Architecture*

Figure 2 outlines the overall architecture of the VITROBENCH test platform. It comprises of the automotive testbed along with relevant hardware and software components. Specifically, the testbed is designed by leveraging the major ECUs of a car that are interconnected to their in-vehicle networks. Meanwhile, the hardware component consists of the equipment required to simulate car functions and the multi-channels/multi-protocols communication board. Finally, the software component is run on a workstation for message sniffing, logging and intercepting the ECU messages.

As illustrated in the leftmost part of Figure 2, real ECUs in our ① *Automotive Testbed* are wired together to create a realistic CAN, CAN-FD and LIN In-Vehicle Network (IVN). Then, protocol messages exchanged on each network are exposed through separate bus channels that are wired to the communication board. It is worthwhile to mention that even though our automotive testbed is composed of a set of specific ECUs for a given car model, it can be reconfigured to work with another set of ECUs for different car models. The testbed may have a new set of ECUs from another car that are interconnected via their IVNs. Additionally, such ECUs will be using the same hardware in VITROBENCH to input simulated signals and communicate with the workstation.

The ② *Hardware* component (see Figure 2) consists of equipment that are needed for the car simulation and the communication board. For example, the car signal simulators send digital or analog car signals to the testbed. The simulated signals are steering angle, speed, signals related to engine such as engine crankshaft/camshaft, and signals from sensors (see Figure 3). The multi channels communication board caters for CAN, CAN-FD and LIN protocols. The sniffed messages for the respective protocol are sent to the workstation embodied in the VITROBENCH test platform for monitoring, logging or modification (e.g., during automated fuzzing of ECUs).

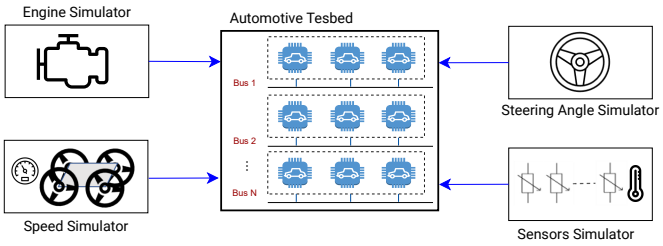

Fig. 3: Simulated Engine, Sensors, Speed and Steering Angle

Messages from the targeted ECU can be captured and modified upon ③a *Interception* (see Figure 2). An ECU can be disconnected from the main network for isolation and modification of messages. Such a feature is crucial for having full control of the IVN traffic. The control of IVN traffic, in turn, allows us to arbitrarily fuzz different ECUs in the IVN as well as to generate targeted attack scenarios. We built a bridge from the ECU to the workstation and back to the main network (see Figure 4). The intercepted messages from the isolated ECU can be modified by the workstation before sending back to the network and vice versa.

In summary, the ③ *software* component in VITROBENCH contains the testing programs (e.g., the fuzzing), testbed communication driver and configuration files that reside in the

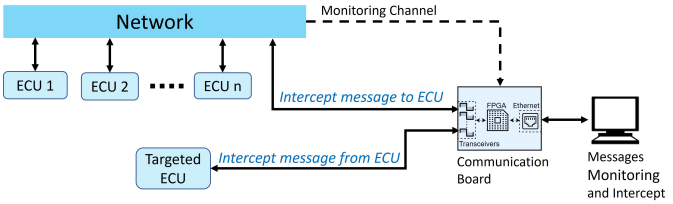

Fig. 4: Message Interception

workstation. These components perform sniffing and logging, frame injection, interception, fuzzing and targeted attacks.

2.2 VITROBENCH *Design Requirements*

Our objective is to make the VITROBENCH test platform an open system that caters to different car architectures and communication protocols. For example, VITROBENCH should be capable to monitor the car messages and simulated signals. Additionally, test cases need to be repeated and test results should be recorded for reproducibility and analysis. Finally, we intend to target arbitrary ECU in the test platform and intercept messages for such ECU. Such capability is required to fingerprint the ECU behavior or to fuzz the respective unit. Based on the aforementioned objective, we outline the following operational requirements for our test platform.

Requirements for Automotive Testbed: The ECUs should have modules for engine control, car access, gateway, motion control, controller for doors, windows and light, steering, fuel control and instrument display. We choose these set of ECUs as these units allow us to create meaningful test scenarios for automotive systems and subsequently, to investigate the impact of the respective test cases. Any targeted ECU can be disconnected from the main network for bypassing the messages via the workstation and sent the messages back (possibly after modifications) to the main network. The diagnostic tool used in our test platform should handle different car models in retrieving the ECU status. This is required for making our testbed general and replaceable by other (compatible) ECUs for a wide variety of car models. Finally, we require the testbed extensible by adding other (compatible) ECUs that can communicate via the considered IVNs in the testbed.

Requirements for Communication Board: Our simulated car has multiple CAN IVNs, one of which is used for ECUs diagnostic. Since we need to perform bridging and injection in multiple IVNs, we require a communication board that can provide access to such IVNs through as many pairs of communication channels as the automotive testbed requires.

Since the communication board can only have a limited number of channels (i.e., eight) before losing its practical size to be easily moved around the testbed, support to cascading multiple boards for future channels expansion is required. Furthermore, the communication board needs to be flexible enough to allow communication with IVNs that employ different automotive protocols other than CAN. To this end, each channel must independently be able to support protocols such as CAN, CAN-FD and LIN with common bitrate[1] requirements as shown in Table I.

Protocol	Bitrate
CAN	100kbps to 1Mbps
CAN-FD	2Mbps to 8Mbpbs
LIN	up to 20kbps

TABLE I: Bitrate requirement for the communication board

Requirements for Car Signal Simulators: To account for a variety of engines in our test platform, we require the engine simulator to provide simulated signals from different engine types. Specifically, the RPM of crankshaft can be varied with the simulator's camshaft RPM changes accordingly. Additionally, the speed simulator can provide different speed from stationary car to speed of at least 50km/h. Such a speed variation allows us to create diverse test scenarios that involve the car in stationary as well as in moving states. Concurrently, the steering angle simulator can give a simulated angle of at least $\pm$ 30 degree for cars in general. Such a range of steering angles helps us to create test scenarios with different driving directions. Finally, we require sensor simulators that can provide resistance, current or periodic signal to simulate different sensors' outputs. We simulate sensors such as fuel level, water fluid level, coolant level, outside temperature, brakes, brake fluid and handbrake.

Requirements for Software Component: We aim to design the software component within our VITROBENCH test platform with the following features and requirement:

- *Sniffing & logging:* It should not affect the normal operation of the car function. Additionally, available CAN Bus Database (DBC[2]) files can be loaded for decoding the sniffed messages. All the messages will be stored in a standard format, i.e. Binary Logging File (BLF) format and further packet inspection can be performed.
- *Frame injection:* A channel can be used to transmit messages to the targeted network with configurable frame interval.
- *Interception:* The intercepted messages will pass through the workstation in both directions. The processing time in the workstation should be kept minimal.
- *Fuzzing & Attacks:* The software should allow a fuzzing program to be attached to the messages. The program can be changed at the workstation for research and development. Different programs for attacks can be explored by modifying the intercepted or injected messages.

In short, the software component should allow the interception and modification of arbitrary ECU messages. Furthermore, such actions should be performed efficiently so as not to affect the normal car functions on the test platform.

[1]https://www.keysight.com/us/en/assets/7018-06531/flyers/5992-3744.pdf
[2]https://www.kvaser.com/developer-blog/an-introduction-j1939-and-dbc-files/

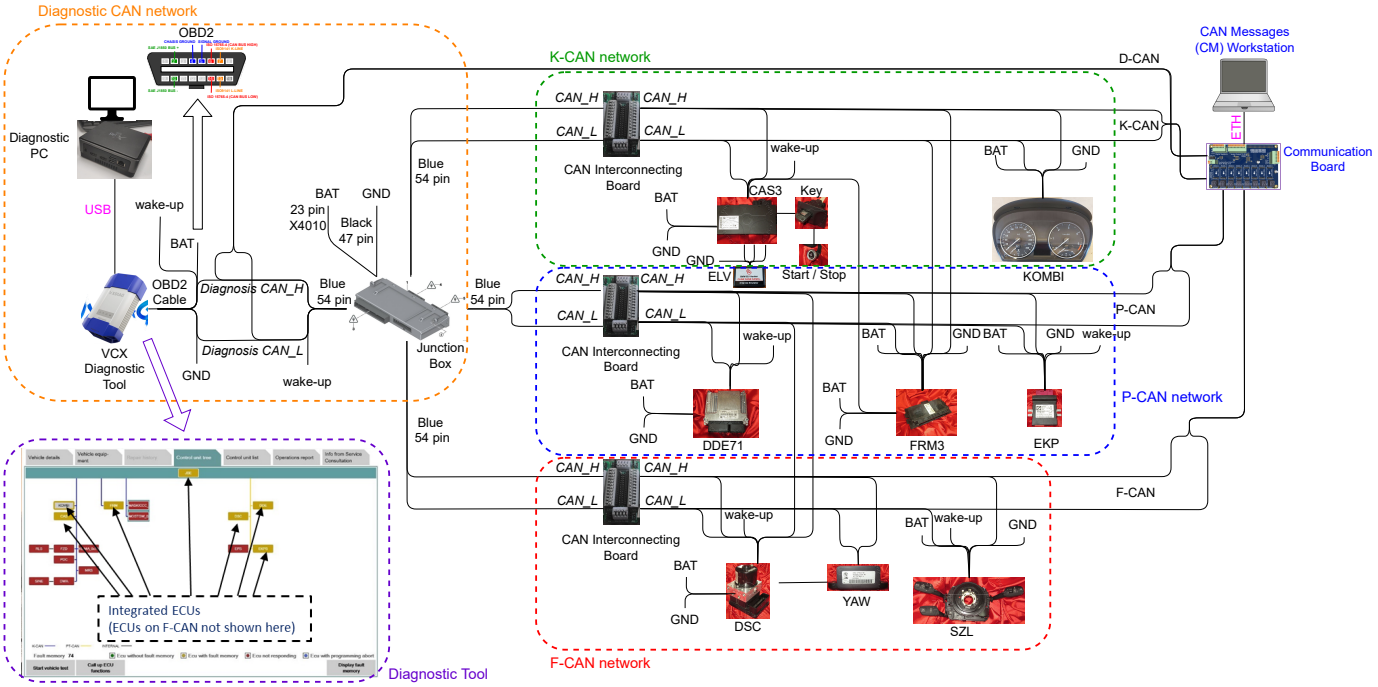

Fig. 5: Automotive Testbed Design

3. Implementation of VitroBench Test Platform

In the following, we detail the implementations of different components in our test platform.

3.1 Automotive Testbed Implementation

For our implementation, the automotive testbed simulates the car model, BMW Series 3 Fifth generation (E90-E93). The testbed consists of major ECUs (see Table II) that are interconnected by their CAN networks as shown in Figure 5. Specifically, the testbed is spread across two test boards, TB1 (Figure 6) and TB2 (Figure 7). TB2 is joined to TB1 on the left by their interconnecting wires. In our testbed, there are three networks, PCAN, FCAN and KCAN, and an external Diagnostic network.

Table II outlines the list of ECUs and their connected network. Appendix A describes the ECUs. The ECUs are purchased from ebay for a total cost of about USD 1500. To know the model of the car and the part number of the ECU, we have leveraged the BMW Parts Catalog[3]

We implement bridging capability in the testbed to intercept messages of a targeted ECU. An example of bridging is shown in Figure 8. In Figure 8, the messages of the engine ECU are isolated from the PCAN network using in-house developed interconnecting boards. The intercepted messages are sent to the workstation and back to the PCAN and vice versa. The intercepted messages can either be pass-through (e.g., for inspection only) or modified by the workstation (e.g., for fuzzing and attacks).

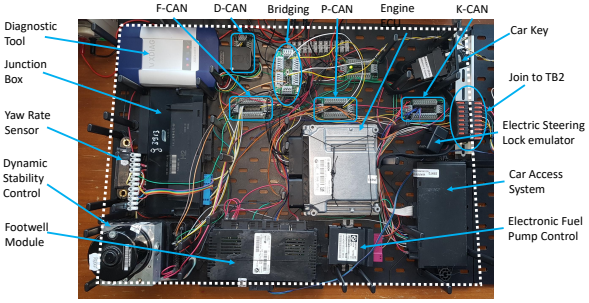

Fig. 6: Testbed Layout - TB1

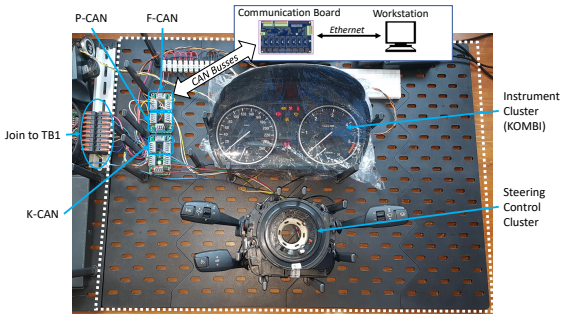

Fig. 7: Testbed Layout - TB2

We select VXDIAG VCX Professional Car Diagnostic[4] as the diagnostic tool. This is because the diagnostic tool supports

[3]https://www.realoem.com/

[4]http://www.allscanner.com/vcx-doip.html

TABLE II: Testbed ECUs and corresponding *X*CAN networks.

Function	ECU	*K*CAN	*P*CAN	*F*CAN
Gateway	JBE	✓	✓	✓
Engine	DDE		✓	
Car Access	CAS	✓		
Doors/Windows/Lights	FRM	✓	✓	
Suspension Stability	DSC		✓	✓
Motion Sensor	YAW			✓
Fuel Control	EKP		✓	
Steering	SZL			✓
Instrument Display	KOMBI	✓		

several car models, including BMW. The tool is connected to the OBD2 interface of the car, i.e., the diagnostic CAN network. It detects the connected ECUs of the testbed and inquires on their status.

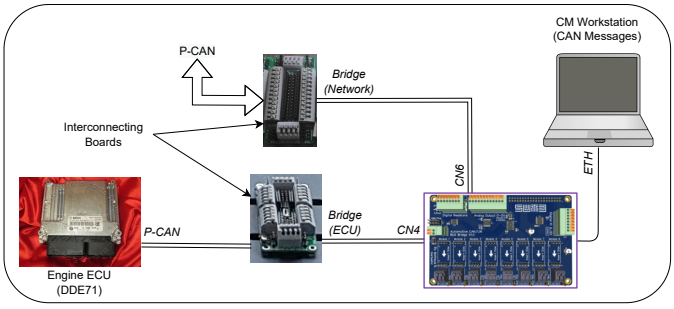

Fig. 8: Bridging the engine ECU (DDE71) from PCAN

3.2 Communication Board and Driver Software

We design the communication board to have eight channels that can be independently configured either as *CAN/CAN-FD* or *LIN* node. Moreover, such channels are accessed from a host machine (workstation) via the Gigabit Ethernet port. Therefore, the user can control more than eight channels by connecting other communication boards to a standard Ethernet Switch in the same network.

Nonetheless, in order to simplify the configuration and usage of the channels within each communication board, we develop a *Driver Software*. Such a driver abstracts the board's channel access over the Ethernet link. This is accomplished by enabling transmission or reception of message frames via multiple *SocketCAN* interfaces, which are accessed through a common socket in Linux operating system.

The overview of the driver software architecture and workflow of frames transmission and reception is illustrated in Figure 9.

As observed in Figure 9, message frames are physically transmitted *(TX)* or received *(RX)* to and from the *testbed*. First, when a frame is received from the *testbed* (see green path in Figure 9), the *Communication Board* forwards the frame to the *Host PC* via Ethernet. Then, the *Driver Software* receives the message frame and broadcasts it to any software (e.g., "Fuzzer software" shown in Figure 9) attached to the correct SocketCAN interface.

Similarly, when any software running on the host machine intends to transmit a message frame to the *testbed* (see blue path in Figure 9), the *Driver Software* receives it and forwards it to the *Communication Board* via Ethernet. In addition, the *Driver Software* exposes UDP sockets to allow Non-Linux software such as TSMaster to receive/send the network frames.

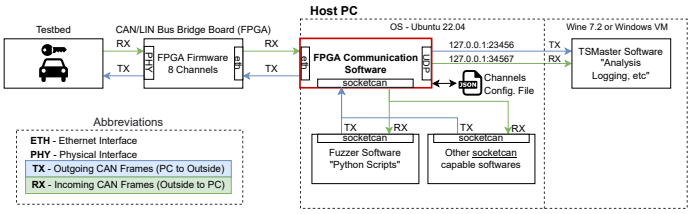

Fig. 9: Communication Board & Driver Software Overview

3.3 Car Signal Simulators

We connect the signal simulators to the automotive testbed, to simulate certain conditions of the car operation. This is illustrated in Figure 10. The testbed is powered from a power supply that provides 13.8V to simulate the car battery. In the following, we discuss the other crucial signals required to operate the testbed.

Engine Simulator: In our test platform, periodic signals and voltages are simulated by the engine simulator, ECU Professional Automobile Signal Simulation, Model MST-9000+. It can simulate several engine types. Additionally, we can program the waveform of engine's rotating crankshaft and camshaft, i.e., RPM. The RPM is varied to simulate different speed of engines in the testbed. The clutch signal is also tapped from one of the 12V outputs from the chosen engine simulator.

Speed and Steering Angle Simulator: We simulate the movement of the car by a rotating gear that are sensed by the wheel speed sensors. The rotating speed of the gear is controlled by a motor controller. In our testbed, the car can be simulated to move from 0km/h to 53km/h. Concurrently, the steering cluster ECU (SZL) provides the steering information. We simulate the steering angle by moving the attached wheel of the steering cluster.

Sensor Simulators: Signals from different car sensors are simulated in our testbed by various resistance and current. They are outlined as follows:

1) *Fuel pump sensing:* We draw 100mA current by attaching resistor to the EKP ECU to simulate the presence of the fuel pump.
2) *Fuel, water fluid, coolant and brake fluid level:* These are simulated by varied resistance for different level.
3) *Outside temperature:* We use varied resistance to simulate measured temperature.
4) *Brake and handbrake:* We use open or closed circuit as activation for the brake and handbrake.

We configure the signal simulators to mimic the car behaviour. For our testing in the laboratory, we fix the input values of the signal simulators. Specifically for each test, we

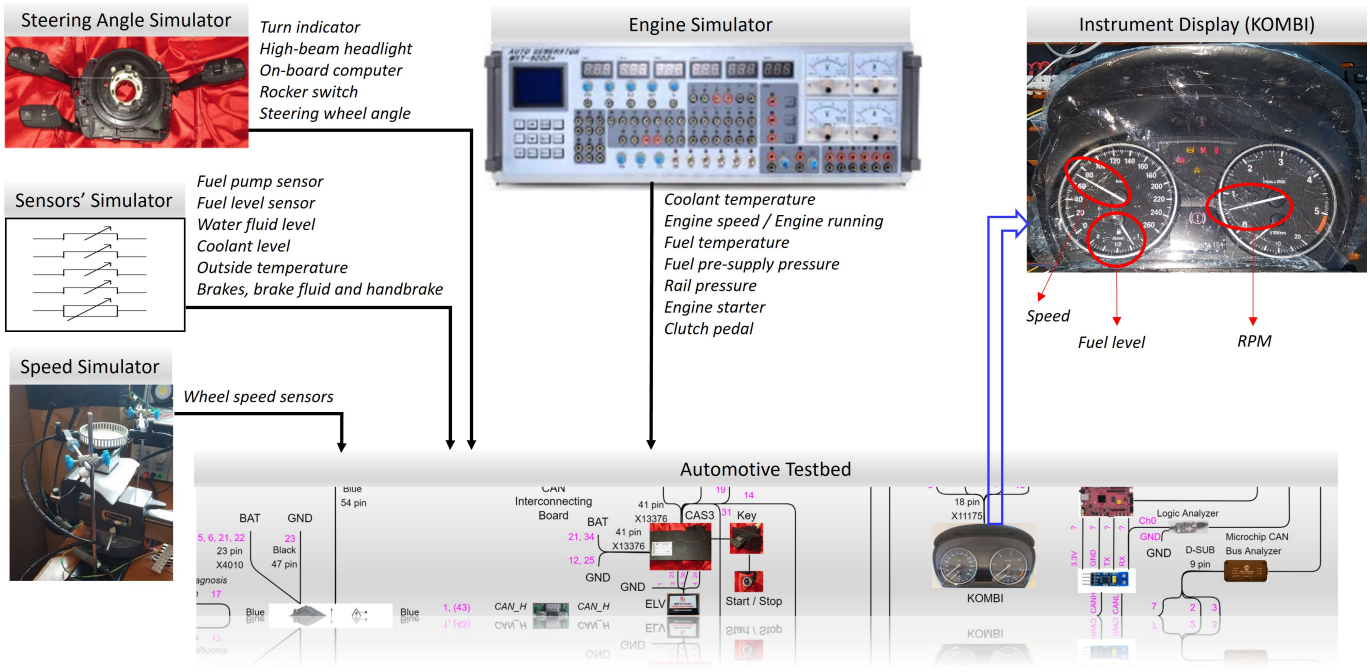

Fig. 10: Car Signal Simulators

focus on the response of certain car's function (e.g., speed display) when we varied our messages for finding vulnerabilities or launching attacks. Even though the car was not driven on the actual road, which will make small changes to signal inputs (for example the RPM due to increased speed), it does not affect our testing. This is because the vulnerabilities appearing due to *CAN messages* still exist despite changes in the input signals. Nonetheless, if the study is on the performance of the car engine or engine stability, then further input modeling might be required to replicate the dynamic environment that accurately captures the changes in input signals.

The input signals can be easily configured for each testing. When setting up, we use the diagnostic tool to check the readings of the signals that are received by the different ECUs. This is then leveraged to approximately create a realistic input signal simulation. Appendix B shows the list of signal inputs that are read by the ECUs. As shown in Appendix B, the signals' RPM, voltage, resistance and open-close circuit are varied to simulate a near-realistic car operating environment.

3.4 Monitoring Signals and Messages

Monitoring all signals and messages are critical to design repeatable tests on our platform. Furthermore, appropriate monitoring allows us to observe the impact on the testbed when an attack is performed. Analog and digital signals are monitored by oscilloscope and logic analyser. The main signals of interest are crankshaft/camshaft RPM, starter motor signal, fuel pump control signal, wakeup signal and CAN messages waveform. The reason for monitoring these signals are as follows. The simulated RPM has to conform to the specific RPM waveform pattern and timing required by the engine ECU in our test platform. Hence, we need to monitor the crankshaft/camshaft RPM. For the starter motor signal, its activation confirms that the function of starting the car works on our testbed. Subsequently, the activation of fuel pump control and wakeup signal also takes place upon starting the car. Finally, the CAN messages waveform is used only for troubleshooting, specifically, to detect any error at any stage of the startup.

Sniffing, injection or interception, and fuzzing or attacks are executed on the CAN Messages (CM) workstation. Three displays monitor the respective messages: 1) *Sniffing and Logging:* When the testbed is operated, all messages from different networks are sniffed and recorded to a logging file. 2) *Injection or Interception Monitoring:* For such monitoring, the display can also be customized for dedicated monitoring channels and parameters. 3) *Fuzzing or attacks:* This display is for monitoring the progress of fuzzing or targeted attacks. The fuzzing or attacks can be launched as python programs or any other software.

3.5 Implementation of Software Component

For sniffing and logging of messages, we leverage TSMaster[5] – an open environment for automotive bus monitoring, testing and logging. The TSMaster display can be customized with graphical meters for speed and RPM, real time parameter values, message traces, decoding using DBC file, and saving to log files of BLF format.

We perform frame injection via a virtual SocketCAN interface, which is available in Linux. This interface is then accessed like any TCP/UDP socket from Python or other

[5]https://github.com/TOSUN-Shanghai/TSMaster

software such as Wireshark, SavvyCan, etc. When write operations are performed on this interface, the communication board receives a request from the host and injects frames to the car IVN. Likewise, interception and bridging is facilitated through software by performing a SocketCAN read and write operation in different CAN channels as discussed in Section 3-B.

We implement fuzzing via python programs. Nonetheless, any programming language can be chosen for this purpose. At present, our testbed includes *random fuzzing* of frame bytes or decoded fields. In particular, after intercepting a frame, we attempt to decode the fields. For the decoded fields, a randomly chosen value is used in the respective before sending the frame back to the IVN. For the bytes that cannot be decoded, for example, due to incomplete DBC file, we randomly select a byte and replace its value randomly, before sending the fuzzed bytes to the IVN.

Although our current implementation only supports random fuzzing, our test platform allows the community to research and implement sophisticated fuzzing algorithms on the test platform. One such approach could be systematic or directed fuzzing to maximize the probability of revealing security bugs (e.g., ECU crashes). For reproducibility and further research, we make the source code of our software component, including the fuzzer source, available in the Appendix C.

4. Evaluation

In this section, we discuss possible usage scenarios of our VitroBench test platform. We first briefly discuss the experimental setup and then discuss the key capabilities of the test platform. Finally, we show three case studies using the test platform in line with security evaluation for automotive systems.

4.1 Testing Setup

There are three possible testing scenarios as illustrated in Figure 11). For normal operation, the four channels of PCAN, FCAN, KCAN and Diagnostic are set up for ① *Sniffing*. For ② *Frame Injection*, an additional injection channel is connected to the network. For bridging, a targeted ECU is separated and re-connected to the ECU channel and back via the network channel as shown in ③ *Interception*.

After connecting the communication channels according to the test scenario, the workstation establishes the ④ *Testing Configuration* as follows:

1) Configures the parameters of each channel by modifying the Channels Configuration File, which is read by the Communication Board.
2) Executes TSMaster to monitor and log messages from all channels.
3) Executes the programs for the different test scenario - Python programs are written for bridging and fuzzing. The programs are using scapy CANSocket for receiving and transmitting CAN messages. Snippets of such a python program is in Appendix C.
4) When the testbed is powered up, messages flow in different CAN networks. These messages are sniffed and displayed on the CM workstation. Data collection is also activated by saving the incoming messages in a log file.

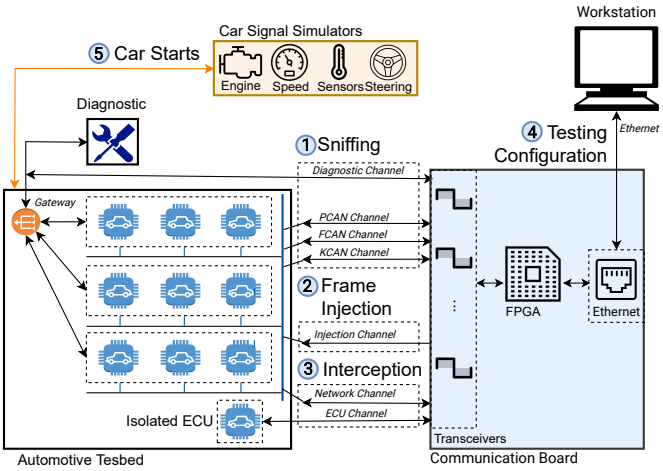

Fig. 11: Testing Scenarios

After test configuration, the ⑤ *Car Starts* by inserting the car key, activating the clutch signal, pressing the "Start button" and activating the crankshaft RPM. The instrument cluster, "KOMBI" (see Figure 7) then displays the engine speed (RPM), fuel level and the car speed from the running testbed. Next, we describe capabilities of this running testbed.

4.2 Testbed Capabilities

We evaluate the three capabilities of the test platform, i.e., sniffing, frame injection and interception. All the IVNs messages are monitored and captured via TSMaster.

4.2.1 Sniffing: There is a plethora of automotive tools for sniffing and analysing CAN/CAN-FD/LIN messages across both Linux and Windows platforms. However, the standardized SocketCAN interface is not supported in many feature-rich free tools such as TSMaster due to the lack of cross-platform compatibility. To have a work-around of this compatibility issue and enable sniffing through other non-standard automotive software, we implement a custom UDP interface to the FPGA communication software. This allows us to successfully sniff and analyse real-time CAN/CAN-FD/LIN messages from TSMaster. For example, Figure 12 illustrates decoded CAN messages in real-time. In this context, the testbed is configured as testing scenario 1 (see Figure 11) and a separate filtered trace window is added for the diagnostic messages.

Concretely, when we vary the RPM, wheels' speed and steering angle; the messages, meters and numerical values change accordingly, as illustrated in Figure 12. The changes can also be observed on the instrument display (see Figure 10). Concurrently, we capture all the messages into a logged .blf file. Such a log file can be played back later in TSMaster to reproduce the test results.

4.2.2 Frame Injection: For testing the capability of frame injection (see test scenario 2 in Figure 11), we inject a fix message ID 0x080 of eight data bytes into PCAN. The

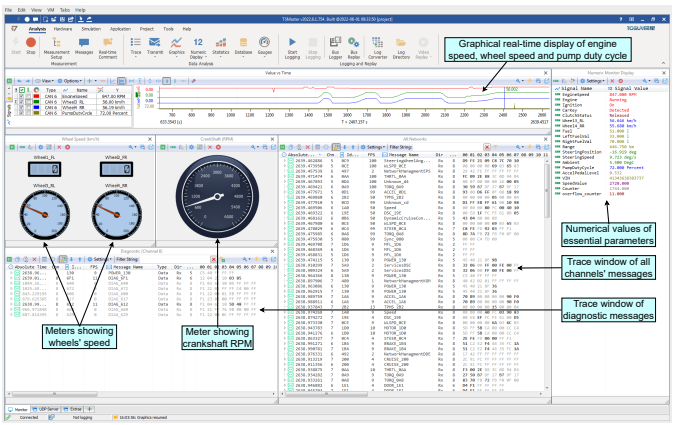

Fig. 12: An Illustration of Sniffing

message is injected using channel 4 of the communication board at interval of 100ms, 30ms, 10ms and 1ms. We sniff the messages flowing in PCAN by channel 6. The results in Figure 13 and Table III show that the injected frames follow the intended time interval with some missed messages that are transmitted at the next interval. The percentage of these missed messages is less than 1%.

TABLE III: Frame injection interval

Frame Interval	100ms	30ms	10ms	1ms	0.1ms
Missed Msgs (%)	0.66	0.56	0.72	0.54	
Average (µs)	100659.5	30167.49	10071.53	1005.269	300.4734
Median (µs)	99999	30000	10000	999	268
Min (µs)	99757	29732	8560	723	168
Max (µs)	200039	60026	20106	3037	549

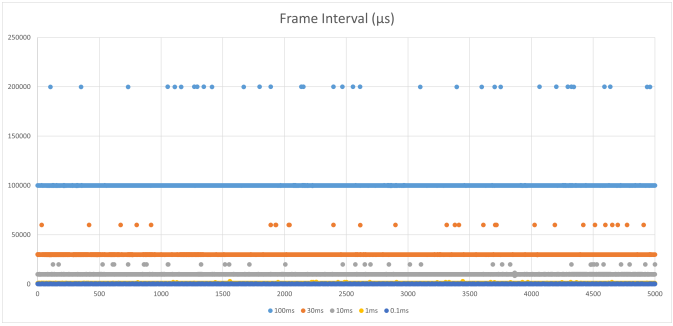

Fig. 13: An illustration of frame injection at different intervals

We also perform injection for an interval of 0.1ms that is less than the time for one message frame. These injected frames do not get transmitted at the intended interval of 0.1ms, but are transmitted at interval of one to two frames time of 0.268ms.

4.2.3 Interception: To test the interception scenario (see test scenario 3 in Figure 11), we isolate the engine ECU by bridging as shown in Figure 8. The intercepted messages of the engine ECU pass through the workstation, but the ECU is connected to channel four of the communication board, whereas the PCAN network is still connected to channel 6. Therefore, we wrote a python program to receive messages from channel four and send to channel six, and vice versa. To measure the effectiveness of interception, we compute the latency of 91,000 messages passing through the workstation. The results are shown in Figure 14. The result shows that all messages are able to pass through the workstation with a reasonable latency i.e., with a median latency of 0.077ms.

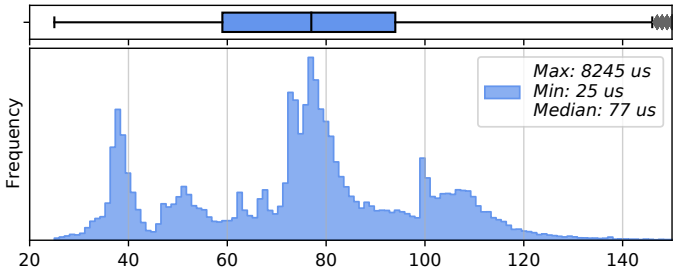

Fig. 14: CAN Frames Pass-through Latency (us)

4.3 Case Studies

In this section, we discuss three different case studies using the sniffing, injection and interception capabilities described in the preceding section.

4.3.1 **Case Study I (Fingerprinting):** We use the VIROBENCH test platform to obtain the behaviour of each ECU and the testbed networks. We call this process *fingerprinting*. Broadly, we perform the following actions to fingerprint arbitrary ECUs:

- Isolate each individual ECU via bridging,
- Collect the CAN messages from the isolated ECU,
- Analyze the collected CAN messages.

From the collected CAN messages, the testbed identifies the ECU's source messages or relay messages if the ECU is connected to multiple networks. In addition, the testbed obtains the statistics of IVN messages from sniffing, such as message ID, its normal inter-frame interval and time deviations. The collected messages and the statistics of IVN messages constitute the fingerprint of the ECUs and the considered car model.

TABLE IV: Number of CAN Messages from ECU

S/N	ECU	Bus	No of Src Msg	No of Relay Msg
1	DSC	P-CAN	13	3
2	DSC	F-CAN	3	1
3	EKP	P-CAN	4	0
4	SZL	F-CAN	4	0
5	CAS	K-CAN	14	0
6	DDE	P-CAN	16	0
7	FRM	P-CAN	0	0
8	FRM	K-CAN	17	0
9	JBE	P-CAN	12	18
10	JBE	K-CAN	24	23
11	KOMBI	K-CAN	23	0
12	YAW	F-CAN	4	0

Table IV captures the statistics of messages collected from each ECU whereas Figure 15 illustrates the frame interval of different messages flowing through PCAN, KCAN and FCAN networks, respectively. To show the reliability of our

fingerprints, we collect the ECU and network messages over three independent experiments spanning two days. We verified that the statistics of messages illustrated in Table IV is exactly the same (hence, deterministic) across the three experiments. Furthermore, we show the inter-frame interval of different messages in Figure 15 over three experiments (three experiments are captured via "4-Jul-1", "5-Jul-2" and "5-Jul-3"). As observed in Figure 15, the inter-frame interval for all messages exhibit negligible variation over different experiments. Hence, our design allows us to reliably fingerprint the network messages and ECUs. Such a fingerprint can be leveraged to detect variation in network behaviour e.g., during a cyber attack.

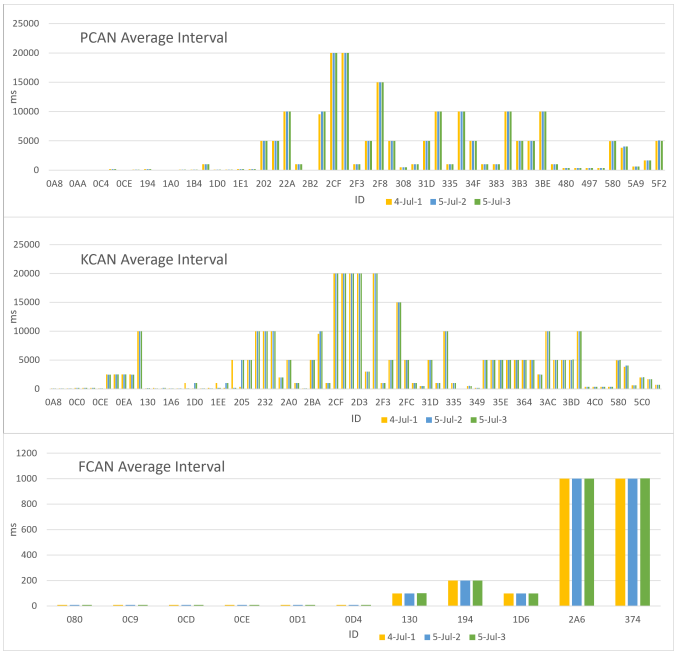

Fig. 15: Inter-frame Interval over three experiments labelled as "4-Jul-1", "5-Jul-2" and "5-Jul-3"

4.3.2 **Case Study II (Random Fuzzing):** Capabilities of our VITROBENCH test platform allow to perform fuzzing arbitrary messages in our testbed. Specifically, once the targeted message is intercepted, there are several ways to fuzz the message to be transmitted back to the main network. For example, we implemented the following fuzzing actions:

1) Fuzz (modify) selected random or targeted message ID,
2) Fuzz selected random or targeted data byte, or
3) Fuzz selected random or targeted message field (for known Whitebox ECU message structure).

Additionally, we implement the following mechanisms to arbitrarily modify the message flow through the network:

1) Block the targeted message,
2) Send the targeted message modified via fuzzing back to network, or
3) Replay multiple copies of the targeted message.

Table V demonstrates the fuzzing on message 0xAA from DDE (engine ECU) in PCAN network. From the testbed bridging setup (see Figure 8), fuzzing is initially performed on

TABLE V: Fuzzing of message `0xAA` from DDE ECU. Fuzzed byte is shown in red.

Time Stamp	ID	Chn	LEN	D1	D2	D3	D4	D5	D6	D7	D8
1349746428	000000AA	4	8	D0	E5	3D	89	26	0A	94	B4
1349746603	000000AA	6	8	D0	E5	3D	89	26	0A	94	62
1349748164	000000AA	4	8	D0	E6	3D	88	26	0A	94	B4
1349748217	000000AA	6	8	D0	E6	3D	88	26	0A	94	6C
1349749189	000000AA	4	8	C2	D7	3D	89	26	0A	94	B4
1349749242	000000AA	6	8	C2	D7	52	89	26	0A	94	B4
1349755443	000000AA	4	8	C3	D8	3D	89	26	0A	94	B4
1349755504	000000AA	6	8	C3	D8	3D	89	26	0A	CF	B4
1349764566	000000AA	4	8	D4	E9	3D	89	26	0A	94	B4
1349764612	000000AA	6	8	D4	E9	4E	89	26	0A	94	B4
1349784969	000000AA	4	8	D5	EA	3D	89	26	0A	94	B4
1349785140	000000AA	6	8	D5	EA	3D	89	26	0A	95	B4
1349786255	000000AA	4	8	D6	EB	3D	89	26	0A	94	B4
1349786273	000000AA	6	8	D6	EB	3D	89	26	0A	58	B4
1349797027	000000AA	4	8	D7	EC	3D	89	26	0A	94	B4
1349797082	000000AA	6	8	D7	EC	3D	89	26	0A	67	B4
1349805462	000000AA	4	8	D7	ED	3D	88	26	0A	94	B4
1349805537	000000AA	6	8	D7	ED	3D	88	26	0A	94	5A

messages related to the fuel pump, i.e., 0xA8, 0xA9, 0xAA and 0x337. Subsequently, fuzzing random bytes of 0xAA is found to have a response on the fuel pump. We note that the message is received from channel four and fuzzed message is transmitted via channel six to PCAN, as shown in Table V. We conduct further targeted testing to focus on 0xAA byte 7 (D8). Specifically, we fuzz D8 to examine the impact on the fuel pump. We discovered that fuzzing 0xAA-D8 causes the pumping signal from EKP, i.e., fuel pump ECU to output an erratic analog signal to the fuel pump motor. We leverage these analysis results obtained from our fuzzing to design and launch a concrete attack i.e., "Fuel pump attack" on the test platform. We discuss our attack scenarios in the next section.

4.3.3 **Case Study III (Attacks):** In this section, we design concrete attacks that are launched in the test platform. For each attack, we also discuss the potential physical impact on the considered car.

Threat Model: For our designed attacks, we assume an attacker who can physically or remotely compromise one or more ECUs and the IVN. Prior work [30] has shown that such a threat is concrete. We consider that the attacker aims to impair or arbitrarily manipulate certain functions of the targeted car. She can accomplish this by modifying certain messages, delaying or dropping messages as well as flooding targeted messages to cause denial of service attacks. We note that our attack model is in line with the adversary models considered in recent works on automotive security [24]. Moreover, considering a strong attacker model that can manipulate messages allows a comprehensive security evaluation.

TABLE VI: Attacks

Message	Attack	Testbed Response
KCAN messages	Message Flooding	Affects instrument cluster display
Message 0xAA	Fuel Pump Attack	Affects the fuel pump function
Message 0x130	Forced Car Stop	Status of car key
Message 0x1A6	Wrong Speed Display	Speed value for instrument cluster
Message 0x600 to 0x6FF	Penetration Test	Range of diagnostic messages

We design our attacks based on the observation made during our fuzzing and fingerprinting, as discussed in the preceding two case studies. Specifically, we observed the response of the messages, e.g., messages shown in Table VI and devise the attack scenarios. Our attack scenarios are illustrated in Figure 16. These attacks are constructed assuming the attackers

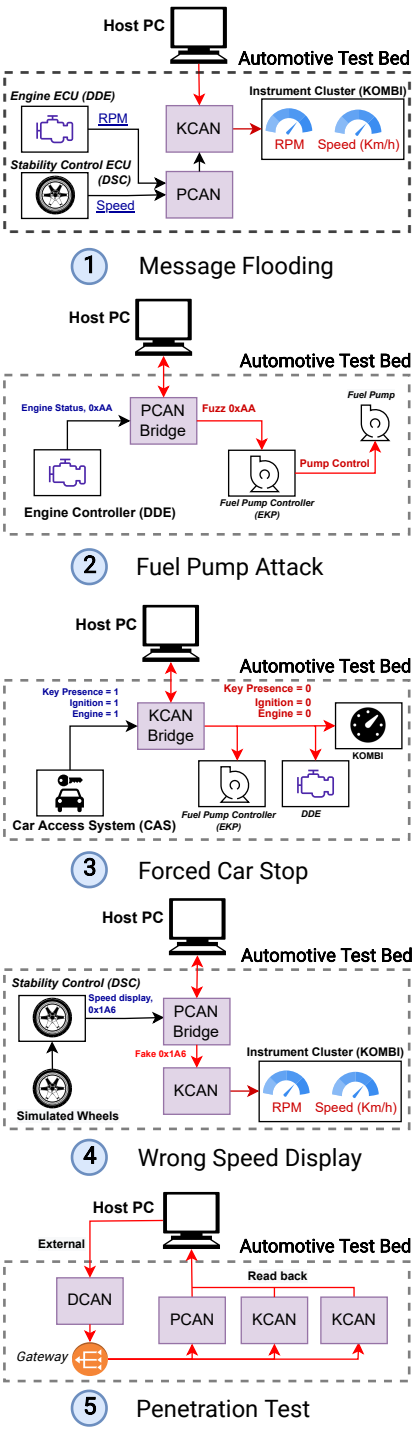

Fig. 16: Attack scenarios on VITROBENCH test platform

are able to maliciously communicate in the IVNs. In line with our threat model, we emulate an attacker-compromised ECU via bridging (see Figure 8). We note that bridging can be used to arbitrarily change messages as well as to inject messages from the workstation. Thus, the bridging capability in our test platform allows us to simulate a connected malicious ECU.

In the following, we detail the concrete attack scenarios illustrated in Figure 16.

① **Message Flooding**: We learn from fingerprinting that display message `0x1A6` flows from DSC ECU PCAN, via JBE gateway to KCAN and finally to Instrument Cluster Display. The objective of this attack is to stop the message `0x1A6` to reach the Instrument Cluster, so as the display in the cluster reflects an incorrect value. To design the attack, we leverage the fact that for CAN protocol, lower message IDs have higher priority than messages with higher IDs. Therefore, we design a flooding attack scenario by flooding messages with IDs lower than `0x1A6` (hence higher priority for transmission). To this end, we design the following exploits for flooding:

- *Exploit 1*: KCAN is connected with an additional injection channel (Figure 11 ②). Then, KCAN is flooded by directly injecting the message ID `0x080` into KCAN by the workstation.
- *Exploit 2*: While exploit 1 floods messages with a single ID, it is also easy to detect via a potential defense e.g., by checking the relative frequency of different message IDs. To make the attack more stealthy, we flood messages with random IDs between `0x080` and `0x1A5`.

We successfully launched the attack and its impact was visible in the Instrument Cluster (see Figure 17). Specifically, the attack stopped the display message to reach the Instrument Cluster. Consequently, the cluster reflected an incorrect value of speed. Such an attack could severely impair the car function. For example, it is possible that the car speed is over the safe driving speed and the Instrument Cluster does not display the same due to an ongoing attack. This, in turn, may result in serious consequences.

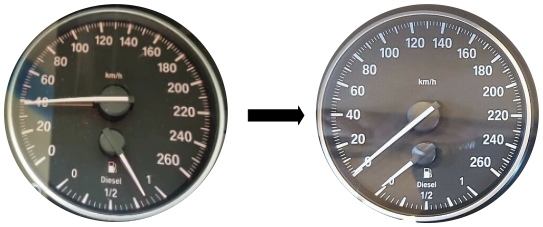

Fig. 17: Message Flooding

② **Fuel Pump Attack**: As discussed in our fuzzing case study, engine status message `0xAA` from DDE is found to affect the pump control signal of EKP. From fingerprinting case study, we observed that EKP message `0x335` reflects the function of the control signal to the fuel pump. Based on this knowledge, we design the fuel pump attack to impair the fuel pump function of the car as follows:

- *Exploit 1*: DDE is isolated and bridged from PCAN (Figure 11 ③). The intercepted message `0xAA` is modified by fuzzing before it is sent to EKP via PCAN.
- *Exploit 2*: PCAN is connected with an additional injection channel (Figure 11 ②). In addition to the normal `0xAA`, we directly inject fuzzed message `0xAA` into PCAN (and received by EKP).

In both exploits, the physical impact is that the fuel is pumped in an irregular fashion to the engine. Concretely, before the attack, the normal response of the analog control signal to the fuel pump motor is observed on an oscilloscope to have 72% duty cycle at steady state (see Figure 18). We note that the message, `0x335-D8` (data byte eight) is found to reflect the duty cycle of the fuel pump control signal.

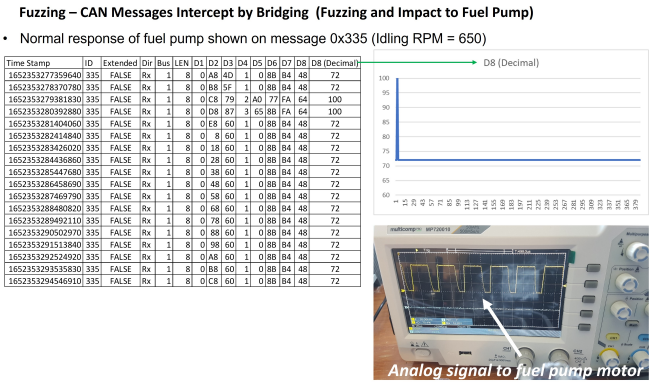

Fig. 18: Before Fuzzing 0xAA

After fuzzing 0xAA D8, the oscilloscope shows the erratic fuel pump signal (see Figure 19). Specifically, the EKP ECU outputs a series of random duty cycle messages as shown on the graph of `0x335-D8` in Figure 20.

Since this attack impairs the fuel pump functionality of the car, this may potentially impact the engine to behave in an erratic fashion during driving. Due to this, drivers may experience sporadic engine behaviors including the engine being stopped while the attack takes place.

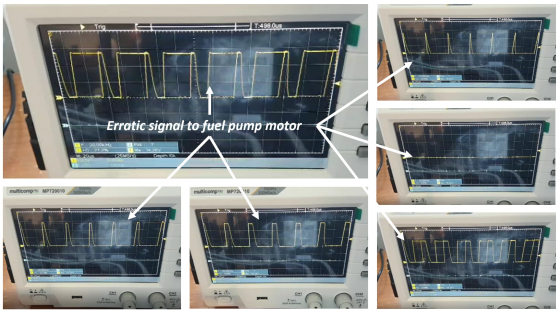

Fig. 19: Erratic Fuel Pump Analog Signal

③ **Forced car stop**: From our fingerprinting case study, we observed that power status message `0x130` flows from CAS ECU over KCAN network. Furthermore, from the DBC decoding file and leveraging bridging, we observed that message `0x130` has three fields: *CarKey*, *Ignition* and *Engine* that affect the engine function.

Based on the information mentioned in the preceding paragraph, we design an attack to force the car to stop. We intercept CAS message `0x130` and farbicate message fields i.e., *CarKey*, *Ignition* and *Engine*. We carry out the attack via the following two exploits.

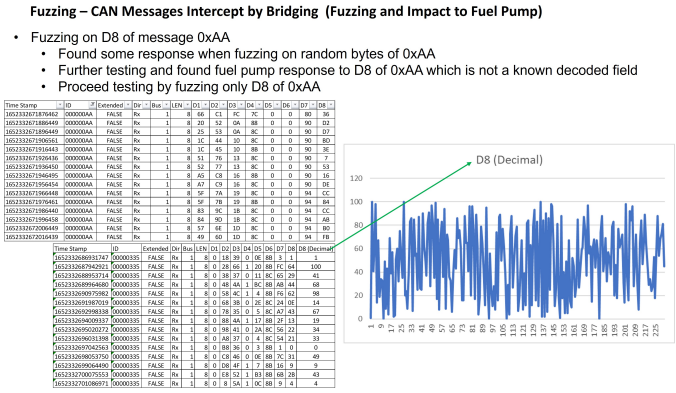

Fig. 20: After Fuzzing 0xAA D8

- *Exploit 1*: CAS is isolated and bridged from KCAN ((Figure 11 ③)). We send the modified message ID `0x130` with *CarKey*, *Ignition* and *Engine* set to zero to KCAN when driver starts the car for the third time.
- *Exploit 2*: Similar to exploit 1, we send the modified message ID `0x130` with *CarKey*, *Ignition* and *Engine* set to zero to KCAN when the car is running and a specified time duration is reached.

In both the exploits, when the attack condition is ongoing, the fuel pump stops. This means the car eventually stops due to the attack. We validate the impact of this attack by observing the 0-RPM speed displayed in the Instrument Cluster of our test platform (see Figure 21). This happens as the engine stops rotation and such is correctly reflected in the Instrument Cluster display. We note that such an attack may have serious consequences in real life. For example, drivers may experience the car being suddenly stopped when the attack takes place.

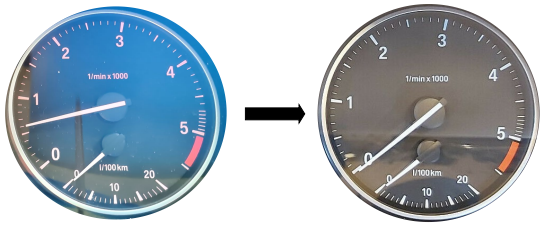

Fig. 21: Forced Car Stop

④ **Instrument Cluster Wrong Speed Display**: From our fingerprinting case study, we observed that the wheel speed message `0xCE` and the display message `0x1A6` flow from the DSC ECU over PCAN network. We employed reverse engineering with the intention to find any correlation between the content in message `0xCE` and message `0x1A6`. We discovered that the wheel speed is encoded in message `0x1A6` transmitted to the Instrument Cluster. Furthermore, we investigated the DBC decoding file to extract the wheel speed from message `0xCE`. Finally, we inspect the relation between this extracted wheel speed and the encoded speed in message `0x1A6`. This, in turn, allows us to decode message `0x1A6` for extracting

the original speed intended to be displayed in the Instrument Cluster.

Based on the study in the preceding paragraph, we design an attack to send fake speed information in message `0x1A6`. Specifically, we design the following exploits to realize the attack scenario:

- *Exploit 1*: DSC is isolated and bridged from PCAN (Figure 11 ③). The car runs normally till the targeted speed for the attack, e.g., 90km/h. We intercept and modify `0x1A6` messages with 2/3 speed. Thereafter, the speed message is incremented by 2/3 of the actual increased speed and clipped at the maximum speed, e.g., 140km/h. This fools the driver that he drives at normal speed even though he accelerates further, and the actual speed may be increased from 90km/h to dangerously 165+ km/h. Due to maximum speed of the simulated wheel motor being less than 53km/h, the attack speed on the testbed is set at 10km/h and clipped at maximum speed of 35km/h.
- *Exploit 2*: Similar to exploit 1, the car runs normally till the attack speed, e.g., 90km/h. Thereafter, the speed in message `0x1A6` is randomly modified. The exploit is launched by intercepting and modifying `0x1A6` messages with random speed value. Due to the maximum speed limit of simulated wheel motor, the attack speed on the testbed is set at 40km/h.

For both the exploits, the attack is successfully realized, as the Instrument Cluster reflects the incorrect speed (see Figure 22). Like the "Flooding Attack" described earlier in this section, this attack may also lead to serious consequences. Specifically, during the attack, the Instrument Cluster provides incorrect information to the driver about the current speed of the car. As a result, the driver may accelerate the car beyond the imposed speed limit. This may not only result in accidents but may also have legal implications.

Fig. 22: Added Random Speed

⑤ **Penetration Test**: The diagnostic CAN network (DCAN) in our test platform is accessible via external OBD2 connection. In this scenario, we conduct penetration test to find message IDs that can pass through the JBE gateway via OBD2 connection. To this end, we generate random messages with IDs from `0x0` to `0x6FF` and the message content is filled with random byte values via fuzzing. These messages are then transmitted from the workstation to the DCAN network. To observe the impact of the attack and discover messages that pass through the gateway, we monitor the PCAN, KCAN and FCAN network.

We observed that no message with IDs between `0x0` and `0x5FF` infiltrates into internal CAN networks (from DCAN). Nonetheless, some messages with IDs between `0x600` and `0x6FF` were found to infiltrate into the internal CAN networks. An example of the message ID (`0x608`) that passes though the JBE gateway is shown in Figure 23. Specifically, we can verify in our test platform that message `0x608` infiltrates from DCAN (Bus=4) to PCAN (Bus=0) network.

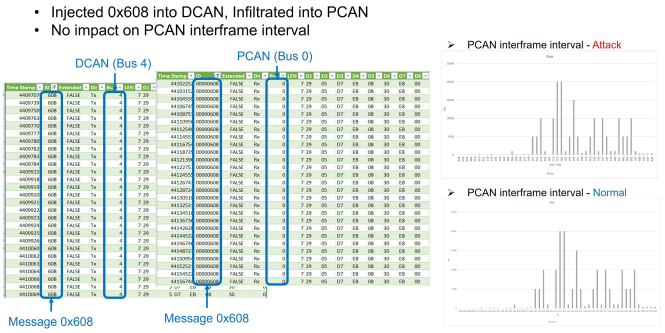

Fig. 23: Impact of Penetration Testing on PCAN

We conducted further investigation to discover any pattern on the messages that pass through the JBE gateway. Our investigation reveals that messages that do not follow the ISO-TP structure [25] are dropped by the gateway. Such an ISO-TP structure uses the D2 byte to include the number of data bytes that follow. We use this knowledge to make our penetration more targeted. Concretely, we only inject messages to the DCAN that follows ISO-TP structure. This allows us to perform more effective penetration testing. Future works on fuzzing may focus on finding these patterns automatically to perform more directed fuzzing actions.

After making the penetration testing targeted and ensuring all fuzzed messages indeed in-filtrate into internal CAN network, we check the impact on the inter-frame interval timing of normal messages communicated within the network. We observed that such penetration testing has little to no impact on the inter-frame interval (see the inter-frame timing distribution in Figure 23). This happens due to the very low priority of messages that actually pass through the JBE gateway and infiltrate into internal CAN network.

4.4 Extension to CAN-FD Network

To show the extensibility and flexibility of our VITROBENCH test platform, we augmented our automotive testbed with COTS ECUs supporting CAN-FD protocol. In the following, we outline some salient aspects of this extension. We also discuss adoption of our key testbed capabilities and use cases (e.g., bridging and fingerprinting) for a different IVN protocol (i.e., CAN-FD).

Bridging: We extended VITROBENCH with two CAN-FD ECUs, Front Radar and Front View Camera, from Kia Sorento. These ECUs are connected with two CAN-FD networks, E-CAN and LOCAL-CAN. Like CAN networks (Figure 8), we also implement the bridging capability in such CAN-FD

network to facilitate interception and fuzzing. The implementation of this bridging is illustrated in Figure 24.

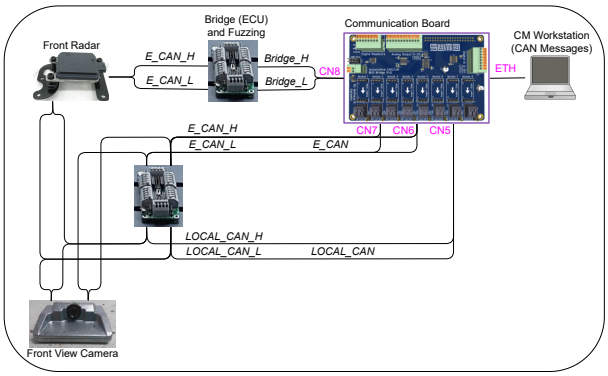

Fig. 24: Bridging of Front Radar CAN-FD ECU for Fuzzing

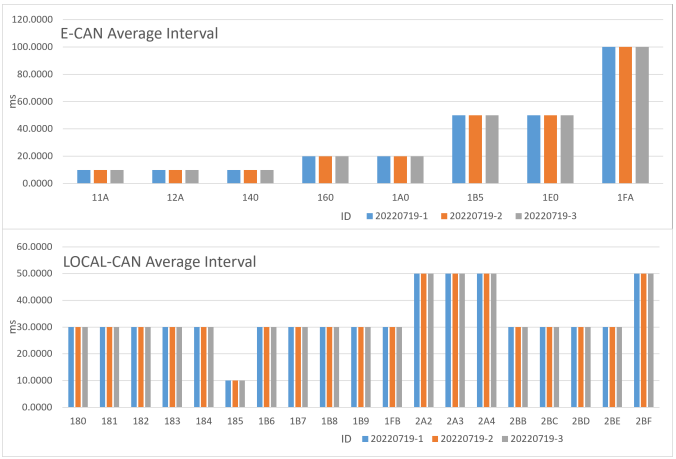

Fig. 25: CAN-FD Frame Interval

Since our communication board supports multiple channels and protocols, including CAN-FD, such an extension of the bridging capability was feasible. Moreover, we leverage the same fuzzing program (see Appendix C) to fuzz arbitrary bytes of arbitrary CAN-FD frames. This shows that bridging is a generic capability that can be used for intercepting and fuzzing multiple IVNs.

Fingerprinting: We employ fingerprinting in terms of reliability collecting the ECU messages and computing the inter-frame interval for each CAN-FD message ID. To validate the reliability of our fingerprinting, we perform three independent experiments to collect the ECU messages and to compute the inter-frame interval. Table VII captures the number of messages from each ECU, as obtained deterministically across three different experiments. Finally, Figure 25) illustrates the inter-frame interval of CAN-FD frames in different CAN-FD networks. Each bar for a given message ID captures an independent experiment. As shown in Figure 25), the variance in the computed inter-frame interval is negligible.

TABLE VII: Number of Messages from Source CAN-FD ECU

S/N	ECU	Bus	No of Src Msg
1	Front Radar	E-CAN	3
2	Front Radar	LOCAL-CAN	1
3	Front View Camera	E-CAN	5
4	Front View Camera	LOCAL-CAN	18

Our extension with CAN-FD ECUs and networks shows that our ideas on bridging and its usage to fingerprint are not limited to CAN networks. These are generic concepts that have the potential to be applied on a larger, versatile and emerging IVNs. Therefore, we believe that our VITROBENCH test platform provides a foundation and the required capabilities for impactful research in automotive cybersecurity.

5. RELATED WORK

Different types of simulation and testbed [29] [26] [9] have been used by researchers for studying different aspects of IVNs and automotive cybersecurity, and most types do not incorporate the vehicle's ECUs. Koscher et al. [31] used vehicles as testbed and with a CAN simulator accessed by OBD scanners. They demonstrated that any compromised ECU can cause adversarial effect to the automotive functions. They purchased two vehicles for testing in the lab and in road tests. Most of their experiments were conducted with the vehicle being stationary. For moving vehicle, they obtain access to the runway of a decommissioned airport to re-evaluate many of the attacks. This work only considered attacks via the OBD-II connector i.e., the external port. In contrast, we provide a platform to investigate and launch attacks in IVNs. Moreover, the focus of the work by Koscher et al. [31] is to show the attack surfaces in modern automobiles, instead of providing a flexible, realistic test platform for researching attacks and defense on automotive systems.

Fowler et al. [23] proposed a hardware-in-the-loop testbed for out of vehicle design, development and testing. A vehicle was simulated on a testbed by using Vector simulator, CANoe software and a Bluetooth-enabled dongle which was connected to the testbed's OBD port. CAN messages were then injected in an undesirable manner to perform a cyber attack on the vehicle and then the simulator. In contrast to our work, the testbed offered by Fowler et al. [23] does not involve any COTS ECUs and the capabilities of the testbed were not discussed or evaluated. Oruganti et al. [16] also proposed a framework of a security evaluation platform based on hardware-in-the-loop. Nonetheless, they only offer ECU simulation and the support to include physical ECUs is planned in future. In contrast, our VITROBENCH test platform provides a foundation for cybersecurity research involving COTS ECUs and IVNs.

Roberts et al. [10] showed that a cyber-physical testbed using MIT CSAIL Duckietown can support a real-world, operational AV shuttle. The testbed that consisted of a small-factor driving range and an autonomous vehicle (DuckieBotIt), had been used for evaluating autonomous driving software. Using this testbed, they demonstrated that the results for testing cyber vulnerabilities can be applied to the iseAuto, an AV shuttle operating in Tallinn, Estonia. The objective of

our work is orthogonal to the objective of Roberts et al. [10]. Specifically, the work by Roberts et al. [10] focuses on finding the vulnerabilities in autonomous driving *software*, instead of investigating the security issues in IVNs. Specifically, the tests considered by Roberts et al. are contexts and scenarios sensed by an autonomous car e.g., road markings. In contrast, the tests in our VITROBENCH platform consists of raw frames communicated within IVNs.

A commercial product, "Portable Automotive Security Testbed with Adaptability" (PASTA) had also been developed by Toyama et al. [22]. PASTA was an open and adaptable platform in one attaché case that consisted of white-box simulated ECUs. It was developed for evaluating vehicle cybersecurity technology and was able to visualise various car internal operations. Similar to PASTA, another testbed Resistant Automotive Miniature Network (RAMN) was developed by Gay et al. [13]. RAMN is a credit-card sized PCB, which contains four ECUs connected to a CAN bus and was compatible with CAN-FD for automotive testing. In contrast to VITROBENCH, both products did not offer a test platform with COTS ECUs (i.e., the ECUs were also developed by the designers). Moreover, both products were mostly proposed without any comprehensive evaluation. As such, the capability of this platform for cybersecurity research remains unclear.

There is a growing interest in Digital Twin [14] [21] [4] in the automotive industry. A Digital Twin testbed can be defined as a virtual representation of a physical product containing information about the car. This technology has contributed to the car design by drawing useful information from its functions. Shetty [6] had developed a digital twin vehicle model using Toyota Prius as the test vehicle. It was equipped with various sensory equipment such as the GNSS/INS module, cameras, thermal cameras, LIDARs and RADARs to record various scenarios while driving. They demonstrated a cruise control model as an alternative to the power and brake subsystems. Popa et al. [5] had developed laboratory CarTwin models for the in-vehicle network to mimic a real vehicle network. The CarTwin employed seven Infineon TC275 lite kits for the seven nodes communicating on the CAN network. The models show good correlation while comparing the models' outputs with the collected data that were extracted from the vehicle. Our work is orthogonal as Digital Twin testbed focuses on the design and modelling aspect, while our testbed focuses on cybersecurity of the IVNs.

Fuzzing is commonly used in communication protocol testing to detect vulnerabilities. Bayer and Ptok [27] developed a fuzzer and showed that it is efficient to detect faults in automotive ECUs. They believed that fuzzing will have a role in a vehicle's security evaluation process in future. Lee et al. [28] studied the impact to a parked car by fuzzing external CAN packets into the OBD port. They performed the attacks without any in-depth knowledge of the car. During the attacks, the abnormal behavior was monitored through the car's instrumentation panel. Zhang et al. [12] proposes a fuzz testing method called CAN-FT to mine vulnerabilities in CAN bus. Their method employed Generative Adversarial Network (GAN) to generate fuzzed messages and uses adaptive boosting (AdaBoost) to detect the abnormal states of CAN bus. Fowler et al. [19] investigated whether a fuzz test, mentioned in SAE J3061, can be widely applied in a vehicle development process. Their results demonstrated that security tests can use fuzzing before the vehicle were being made for series production. Vinzenz and Oka [11] integrated fuzz testing into the cybersecurity validation strategy. They showed that fuzz testing was beneficial to improve product security by providing inputs to enhance other testing activities.

VITROBENCH is complementary to the aforementioned efforts involved in developing fuzzing methodologies for automotive systems. Specifically, contrary to providing a specific fuzzing methodology, VITROBENCH provides a comprehensive test platform to facilitate realistic evaluation of fuzzing solutions, among others. Nonetheless, during evaluation we also show that our fuzzing guides us to design concrete attack scenarios.

Park et al. [17] proposed a security evaluation methodology and tool based on four types of attacks (denial of service, data frame replay, fuzzing, impersonation) that can be performed on In-vehicle CAN. Another CAN security evaluation tool (CANsec) was proposed by Zhang et al. [15] to add a few other tools such as CarShark (CAN network analysis tool) [31], ATG [20] for attack models and a security evaluation tool [17]. Park et al. [17] also developed an evaluation tool for responses beyond network communication using various sensors such as video, sound and vibration sensors. Similarly, an experimental sensor harness was developed by Werquin el al. [18]. In contrast to VITROBENCH, none of these works propose a test platform involving COTS ECUs, instead these works aim to enhance the automation of automotive cyber security evaluation via attack generation tools and via sensor harness. Thus, our goal is orthogonal to these works.

In summary and to the best of our knowledge, VITROBENCH is the first comprehensive test platform involving COTS ECUs and IVNs that allow researchers to have full control over the internal car network. This facilitates evaluation of arbitrary attacks on IVNs in a realistic fashion despite having access to a car.

6. DISCUSSION AND FUTURE OUTLOOK

In this section, we discuss our outlook to inspire future cybersecurity research for automotive systems using VITROBENCH.

6.1 Towards a flexible test platform

Automotive cybersecurity concerns are increasing over the years [1]. Much work have been performed to test vehicles along different aspects of cybersecurity. However, each vehicle design is different and it is difficult to use a single testbed for all types of vehicles. To address such challenge, we envision a test platform using COTS ECUs in the laboratory. This allows our tests to be repeatable (i.e., consistent), yet the test results to be close to the performance in a real car that embodies the

respective ECUs. More importantly, we carefully design VITROBENCH such that ECUs in the automotive testbed can be replaced (or added/removed) keeping the rest of the platform (e.g., the communication board and software components) untouched. Such a design facilitates an extensible and flexible test platform, while reusing VITROBENCH capabilities and use cases such as bridging, sniffing and fuzzing, among others. As a byproduct of our implementation, the bridging allows to reveal the source and destination ECUs of an intercepted message, which is otherwise impossible via only sniffing the network (e.g., for CAN). This allows us to design targeted attacks for evaluation within VITROBENCH.

Unlike using a real car [31], which is subject to the elements on the road, our laboratory testing is accomplished in a controlled environment and is safe. Test results can also be analysed offline and further testing or changes can be immediately performed for validation in the laboratory. Similarly, unlike using simulated ECUs for testing [23, 10, 22] [10], VITROBENCH offers significantly less ambiguity on the validity of test results on the actual car.

In the future, we aim to extend VITROBENCH for a combination of CAN-FD, Automotive Ethernet and LIN networks. We have designed our communication board for CAN/CAN-FD and LIN, and we will be adding Automotive Ethernet communication capability to our test platform in the future. Indeed, we show in Section 4-D an extension of VITROBENCH for simple CAN-FD network. We demonstrated that the key VITROBENCH capabilities and use cases e.g., bridging and fingerprinting can be easily ported across networks. Having multiple types of IVNs in VITROBENCH will allow researchers to launch and evaluate cross network attacks.

We envision VITROBENCH to be expanded to multiple workstations where each workstation can be connected to a communication board to access the IVNs for different studies. Such studies may include our research in cybersecurity, supporting automotive software development and testing and logging of ECUs and V2V/V2X wireless communication, among others.

6.2 Towards directed fuzzing of automotive networks

In VITROBENCH, we designed a simple random fuzzing algorithm to arbitraily manipulate the messages in IVNs. Despite its simplicity, we show that fuzzing is effective in providing useful information to the designer. For instance, it helped us to design and successfully launch a concrete *Fuel Pump Attack*. The need to use fuzzing for effectively testing automotive systems has also been exemplified by prior works [28, 12]. Nonetheless, we believe that much work is needed in designing effective and directed fuzzing algorithms for automotive systems. While fuzzing technologies have many success stories for testing application software; development and translation of fuzzing technologies for automotive systems is far from mature. Unlike application software, which are easily available as test subjects, having a realistic test platform for automotive systems remains challenging. This, in turn, is detrimental to the progress of security testing technologies for automotive.

Thanks to VITROBENCH, we open the door for development and realistic evaluation of fuzzing research on IVNs. We decouple the design of the automotive testbed and the software component (including fuzzing) within VITROBENCH. This allows the researchers in fuzzing to fully focus on the software component, while keeping the hardware component e.g., the communication board design and the automotive testbed untouched. Future research may involve design of fuzzing algorithms that are directed with the aim to reveal maximum number of vulnerabilities or attacks in the considered IVNs. Another line of research may focus on automatically detecting the abnormal behaviours in the IVNs during fuzzing. While some initial works have been accomplished [12, 17, 18], much work is still needed to automatically detect vulnerable communication traces. Thanks to VITROBENCH, we believe that our *fingerprinting* can provide a useful artifact for detecting abnormal ECU behaviour.

7. CONCLUSION

In this paper, we present VITROBENCH, a configurable test platform using COTS ECUs. VITROBENCH decouples the design of automotive testbed and the accompanying hardware and software. This allows to replace the automotive testbed with different ECUs, while keeping the rest of the VITROBENCH design untouched. Likewise, VITROBENCH may support additional IVN protocols by augmenting the design of the communication board and inspire future research on cybersecurity such as fuzzing by enhancing the software component.

We comprehensively evaluate VITROBENCH to demonstrate its capability. To replicate the test environment in VITROBENCH, we inject signals simulated by external hardware to the automotive testbed. Then, VITROBENCH isolates a targeted ECU via bridging. This allows us to arbitrarily control the traffic within IVNs. This is shown to be effective for intercepting, fuzzing and launching concrete attacks on the platform. Due to the lab environment, all tests within VITROBENCH are repeatable, yet the respective test results are largely realistic due to the embodiment of COTS ECUs within VITROBENCH. Finally, we discuss an extension of VITROBENCH for CAN-FD network to show the generalisability and flexibility. We hope that VITROBENCH provides a foundation for automotive cybersecurity research in future. For replication and research, we have also made our code for the software component available in Appendix C. More information can be obtained from the following website:

https://www.vitrobench.com

ACKNOWLEDGEMENTS

This research/project is supported by the National Research Foundation, Singapore, and Land Transport Authority under Urban Mobility Grand Challenge (UMGC-L011). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Land Transport Authority.

References

[1] Trupil Limbasiya et al. "A systematic survey of attack detection and prevention in Connected and Autonomous Vehicles". In: *Veh. Commun.* 37 (2022), p. 100515.

[2] MathWorks. *MATLAB*. (Accessed 16th December 2022). 2022. URL: https://www.mathworks.com/products/matlab.html/.

[3] MathWorks. *Simulink*. (Accessed 16th December 2022). 2022. URL: https://www.mathworks.com/products/simulink.html.

[4] Dimitrios Piromalis and Antreas Kantaros. "Digital twins in the automotive industry: The road toward physical-digital convergence". In: *Applied System Innovation* 5.4 (2022), p. 65.

[5] Lucian Popa, Adriana Berdich, and Bogdan Groza. "CarTwin—Development of a Digital Twin for a Real-World In-Vehicle CAN Network". In: *Applied Sciences* 13.1 (2022), p. 445.

[6] SS Shetty. "Development of a Digital Twin of a Toyota Prius Mk4". PhD thesis. Eindhoven University of Technology Eindhoven, The Netherlands, 2022.

[7] Vector. *CANoe*. (Accessed 16th December 2022). 2022. URL: https://www.vector.com/int/en/products/products-a-z/software/canoe//.

[8] WIRED. *Hackers Remotely Kill a Jeep on the Highway—With Me in It*. (Accessed 16th December 2022). 2022. URL: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

[9] Youngho An et al. "Design and implementation of a novel testbed for automotive security analysis". In: *Innovative Mobile and Internet Services in Ubiquitous Computing: Proceedings of the 14th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2020)*. Springer. 2021, pp. 234–243.

[10] Andrew Roberts, Olaf Maennel, and Nikita Snetkov. "Cybersecurity Test Range for Autonomous Vehicle Shuttles". In: *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2021, pp. 239–248.

[11] Nico Vinzenz and Dennis Kengo Oka. *Integrating fuzz testing into the cybersecurity validation strategy*. Tech. rep. SAE Technical Paper, 2021.

[12] Haichun Zhang et al. "CAN-FT: A Fuzz Testing Method for Automotive Controller Area Network Bus". In: *2021 International Conference on Computer Information Science and Artificial Intelligence (CISAI)*. 2021, pp. 225–231. DOI: 10.1109/CISAI54367.2021.00050.

[13] Camille Gay, Tsuyoshi Toyama, and Hisashi Oguma. "Resistant Automotive Miniature Network". In: *Proceedings of the Chaos Computer Congress, Leipzig, Germany*. 2020, pp. 27–30.

[14] David Jones et al. "Characterising the Digital Twin: A systematic literature review". In: *CIRP Journal of Manufacturing Science and Technology* 29 (2020), pp. 36–52. ISSN: 1755-5817. DOI: https://doi.org/10.1016/j.cirpj.2020.02.002. URL: https://www.sciencedirect.com/science/article/pii/S1755581720300110.

[15] Haichun Zhang et al. "CANsec: a practical in-vehicle controller area network security evaluation tool". In: *Sensors* 20.17 (2020), p. 4900.

[16] Pradeep Sharma Oruganti, Matt Appel, and Qadeer Ahmed. "Hardware-in-loop based automotive embedded systems cybersecurity evaluation testbed". In: *Proceedings of the ACM Workshop on Automotive Cybersecurity*. 2019, pp. 41–44.

[17] Hyun-Bae Park et al. "Practical Methodology for In-Vehicle CAN Security Evaluation." In: *J. Internet Serv. Inf. Secur.* 9.2 (2019), pp. 42–56.

[18] Timothy Werquin et al. "Automated fuzzing of automotive control units". In: *2019 International Workshop on Secure Internet of Things (SIOT)*. IEEE. 2019, pp. 1–8.

[19] Daniel S. Fowler et al. "Fuzz Testing for Automotive Cyber-Security". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2018, pp. 239–246. DOI: 10.1109/DSN-W.2018.00070.

[20] Tianxiang Huang, Jianying Zhou, and Andrei Bytes. "ATG: An attack traffic generation tool for security testing of in-vehicle CAN bus". In: *Proceedings of the 13th International Conference on Availability, Reliability and Security*. 2018, pp. 1–6.

[21] Rajeeth Tharma, Roland Winter, Martin Eigner, et al. "An approach for the implementation of the digital twin in the automotive wiring harness field". In: *DS 92: Proceedings of the DESIGN 2018 15th International Design Conference*. 2018, pp. 3023–3032.

[22] Tsuyoshi Toyama et al. "PASTA: portable automotive security testbed with adaptability". In: *London, blackhat Europe* (2018).

[23] Daniel S Fowler et al. "Towards a testbed for automotive cybersecurity". In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2017, pp. 540–541.

[24] Kyong-Tak Cho and Kang G. Shin. "Fingerprinting Electronic Control Units for Vehicle Intrusion Detection". In: *USENIX Security Symposium*. USENIX Association, 2016, pp. 911–927.

[25] *ISO 15765-2:2016 - Diagnostic communication over Controller Area Network (DoCAN) — Part 2: Transport protocol and network layer services*. https://www.iso.org/standard/66574.html. 2016.

[26] Shane Tuohy et al. "Hybrid testbed for simulating in-vehicle automotive networks". In: *Simulation Modelling Practice and Theory* 66 (2016), pp. 193–211.

[27] Stephanie Bayer and Alexander Ptok. "Don't fuss about fuzzing: Fuzzing controllers in vehicular networks". In: *13th escar Europe* (2015), p. 88.

[28] Hyeryun Lee et al. "Fuzzing CAN Packets into Automobiles". In: *2015 IEEE 29th International Conference*

on Advanced Information Networking and Applications. 2015, pp. 817–821. DOI: 10.1109/AINA.2015.274.

[29] Christopher E Everett and Damon McCoy. "{OCTANE}(open car testbed and network experiments): bringing cyber-physical security research to researchers and students". In: *6th Workshop on Cyber Security Experimentation and Test ({CSET} 13)*. 2013.

[30] Stephen Checkoway et al. "Comprehensive Experimental Analyses of Automotive Attack Surfaces". In: *USENIX Security Symposium*. USENIX Association, 2011.

[31] Karl Koscher et al. "Experimental security analysis of a modern automobile". In: *2010 IEEE symposium on security and privacy*. IEEE. 2010, pp. 447–462.

APPENDIX A
ECUS DESCRIPTION

Function	ECU	*K*CAN	*P*CAN	*F*CAN	Description
Gateway	JBE	✓	✓	✓	The junction box (JBE) assumes a central role in the vehicle. The junction box electronics is the central gateway in the vehicle.
Engine	DDE		✓		The engine control unit (DDE) controls a series of actuators on an internal combustion engine to ensure optimal engine performance. It does this by reading values from a multitude of sensors within the engine bay, interpreting the data and adjusting the engine actuators.
Car Access	CAS	✓			The car access system (CAS) is an antitheft alarm system and enables the start of BMW vehicles.
Doors/Windows/Lights	FRM	✓	✓		The footwell module (FRM) functions as an electrical hub on the drivers side. The FRM receives signals from the doors, it controls the lighting, it commands the adaptive headlights and it also interfaces with the dashboard.
Suspension Stability	DSC		✓	✓	The dynamic stability control (DSC) is a suspension control system. It works by monitoring each wheel speed individually along with yaw rate and acceleration.
Motion Sensor	YAW			✓	The yaw rate sensor measures the vehicle's angular velocity about its vertical axis in order to determine the orientation of the vehicle.
Fuel Control	EKP		✓		The electronic fuel pump control module (EKP) controls the fuel pump.
Steering	SZL			✓	The steering control unit (SZL) is mounted with various switches, including a wiper, turn signal and cruise control. Its steering angle sensor determines where the driver wants to steer, matching the steering wheel with the vehicle's wheels.
Instrument Display	KOMBI	✓			The KOMBI control unit is the module that controls the instrument cluster. The instrument cluster is located on the dashboard, directly in front of the driver behind the steering wheel. It relays information about the vehicle to the driver through the signals from the gauges and warning lights.

APPENDIX B
SIMULATION READING FROM DIAGNOSTIC TOOL

S/N	ECU	Car Status	Simulation	Reading from Diagnostic Tool
1	DDE	Coolant temperature	2.0 V	45.0 °C
2		Crankshaft RPM	650 Hz	649 RPM
3			Crankshaft waveform	Engine turns
4		Fuel temperature	1.0 V	68.7 °C
5		Fuel pressure	2.0 V	2247 mbar
6		Rail pressure	1.0 V	543.3 bar
7		Clutch signal	12 V / Open Circuit	Operated / Not Operated
8			12 V / Open Circuit	90 % / 10 %
9		Neutral gear	Square Wave	49.46%
10		Battery	13.8 V	13400 mV
11		Brake light signal	Open Circuit / GND	Operated / Not Operated
12		Brake light test signal	Open Circuit / GND	Operated / Not Operated
13	JBE	Fuel tank (Left)	330 Ohm	332.8 Ohm
14		Fuel tank (Right)	1000 Ohm	1042.2 Ohm
15		Washer fluid level	GND	Over Minumum
16		Coolant level	GND	Over Minumum
17	FRM	Left rear indicator	Activate On / Off	On / Off
18		Left front indicator	Activate On / Off	On / Off
19		Right front indicator	Activate On / Off	On / Off
20		Right direction indicator	Activate On / Off	On / Off
21		High beam flasher	Activate On / Off	On / Off
22		High beam	Activate On / Off	On / Off
23	KOMBI	Crankshaft RPM	650 Hz	649 RPM
24		Fuel tank (Left)	330 Ohm	14.88 l
25		Fuel tank (Right)	1000 Ohm	35.00 l
26		Button on KOMBI	Pressed / Not Pressed	Pressed / Not Pressed
27		BC/CC button	Pressed / Not Pressed	Pressed / Not Pressed
28		Outside temperature	Variable resistor	-4.5 to 50.0 °C
29			Variable resistor	-4.5 to 50.0 °C
30		Rocker switch	Rocker Down / Neutral	Pressed / Not Pressed
31			Rocker Up / Neutral	Pressed / Not Pressed
32	CAS	Key	Key In / Out	1 / Ignition key not in the ignition lock
33		VIN number	Power On	WBAPN12010A468877
34		Clutch	12 V / Open Circuit	Depressed / Not Depressed
35	DSC	Brake fluid level	GND	O.K.
36		Brake light	Open Circuit / GND	Operated / Not Operated
37		Parking brake warning	GND / Open Circuit	Operated / Not Operated
38		Wheel speed (Rear Left)	Varies motor speed	0 to 53 km/h
39		Wheel speed (Rear Right)	Varies motor speed	0 to 53 km/h
40	EKP	Current fuel pump	100 Ohm (5W)	0.1 A
41		Voltage fuel pump	100 Ohm (5W)	9.6 V

APPENDIX C
SNIPPETS OF A TYPICAL PYTHON PROGRAM

Libraries:

```
import sys
import signal
from threading import Thread
from scapy.contrib.cansocket import CANSocket
from time import sleep
from random import randint
from can import rc as can_rc
```

Defining the CAN sockets - ecu_socket is for output to the bridged ECU, net_socket is for output to the desired network:

```
# Initialise Linux CAN Socket
e_soc, e_rate = 'can_7', 500000        # E-CAN & bitrate
l_soc, l_rate = 'can_6', 500000        # L-CAN & bitrate

# Bridge parameters
net_can, net_br = e_soc, e_rate
ecu_can, ecu_br = l_soc, net_br
net_socket = CANSocket(channel=net_can, bitrate=net_br, receive_own_messages=False)
ecu_socket = CANSocket(channel=ecu_can, bitrate=ecu_br, receive_own_messages=False)
```

Function for receiving from bridged ECU and fuzzed messages to connected network:

```
def bridge_net():

    while True:

        # ======== in_can =========
        # Receive can frame from ecu
        pkt = ecu_socket.recv()

        # ======== packet info =========
        d = bytearray(pkt.data)
        d_len = len(d)
        sel_d = randint(0, d_len-1)
        d[sel_d] = randint(0, 255)
        pkt.data = d

        # ======== out_can =========
        # Transmit frame to net
        net_socket.send(pkt)
```

Function for receiving from connected network and fuzzed messages to bridged ECU:

```
def bridge_ecu():

    while True:

        # ======== in_can =========
        # Receive can frame from ecu
        pkt = net_socket.recv()

        # ======== packet info =========
        d = bytearray(pkt.data)
        d_len = len(d)
        sel_d = randint(0, d_len-1)
        d[sel_d] = randint(0, 255)
        pkt.data = d

        # ======== out_can =========
        # Transmit frame to net
        ecu_socket.send(pkt)
```

Main program:

```
# *** Define signal handler to stop this program
def signal_handler(sig, frame):
    print('\nYou pressed Ctrl+C!')
    sys.exit(0)

# Install signal for program exit
signal.signal(signal.SIGINT, signal_handler)

try:
    # Start bridge
    Thread(target=bridge_net, daemon=True).start()
    Thread(target=bridge_ecu, daemon=True).start()

    while True:
        sleep(50000)

except KeyboardInterrupt:
    print("\nBridge stopped")
```

Instead of fuzzing random byte, if the message is decoded, the decoded field can be fuzzed. Snippets of fuzzing decoded field are as follows.

```
def field_fuzz(pckt):
    decoded_pkt = SignalHeader(bytes(pckt))
    pkt_fields = decoded_pkt.payload.fields
    fields_len = len(pkt_fields)
    pkt_fields_desc = decoded_pkt.payload.fields_desc
    if fields_len == 0:                  # No field, raw data
        pkt_fields = decoded_pkt.payload.payload.fields
        pkt_fields_desc = decoded_pkt.payload.payload.fields_desc
        fields_len = len(pkt_fields)
    sel_field = randint(0, fields_len-1)
    # sel_field = 7  # targeted field
    fieldname, fmt = pkt_fields_desc[sel_field].name, pkt_fields_desc[sel_field].fmt
    if fieldname == 'load':                  # Selected field = raw data
        d = bytearray(pckt.data)
        d_len = len(d)
        sel_d = randint(0, (d_len-1))
        d[sel_d] = randint(0, 255)
        pckt.data = d
    else:
        if fmt[1] == 'B':                    # unsigned integer
            nbr_dig, nbr_scale = pkt_fields_desc[sel_field].size, pkt_fields_desc[sel_field].scaling
            nbr_min = 0
            nbr_max = (2**nbr_dig) - 1
            rdn_val = int(randint(nbr_min, nbr_max) * nbr_scale)
        elif fmt[1] == 'b':                  # signed integer
            nbr_dig, nbr_scale = pkt_fields_desc[sel_field].size, pkt_fields_desc[sel_field].scaling
            nbr = 2**(nbr_dig-1)
            nbr_min = - nbr
            nbr_max = nbr-1
            rdn_val = int(randint(nbr_min, nbr_max) * nbr_scale)
        elif fmt[1] == 'H':
            nbr_min = 0
            nbr_max = 65535
            rdn_val = randint(nbr_min, nbr_max)
        else:
            nbr_min = 0
            nbr_max = 0
            rdn_val = 0
        pkt_fields[fieldname] = rdn_val
        pckt.data = decoded_pkt.payload

def bridge_net():

    while True:
        # Receive can frame
        pkt = ecu_socket.recv()
        if (pkt.identifier == 0x337):
            field_fuzz(pkt)
```