A Virtual Prototyping Platform for Exploration of Vehicular Electronics

Md Rafiul Kabir, Bhagawat Baanav Yedla Ravi, and Sandip Ray *Senior Member, IEEE*

***Abstract*—A critical requirement for robust, optimized, and secure design of vehicular systems is the ability to do system-level exploration, *i.e.*, comprehend the interactions involved among ECUs, sensors, and communication interfaces in realizing system-level use cases and the impact of various design choices on these interactions. This must be done early in the system design to enable the designer to make optimal design choices without requiring a cost-prohibitive design overhaul. In this paper, we develop a virtual prototyping environment for the modeling and simulation of vehicular systems. Our solution, VIVE, is modular and configurable, allowing the user to conveniently introduce new system-level use cases. Unlike other related simulation environments, our platform emphasizes coordination and communication among various vehicular components and just the abstraction of the necessary computation of each electronic control unit. We discuss the ability of VIVE to explore the interactions between a number of realistic use cases in the automotive domain. We demonstrate the utility of the platform, in particular, to create real-time in-vehicle communication optimizers for various optimization targets. We also show how to use such a prototyping environment to explore vehicular security compromises. Furthermore, we showcase the experimental integration and validation of the platform with a hardware setup in a real-time scenario.**

***Index Terms*—Virtual Prototyping, Digital Twin, Automotive, Electronics, Software Process, Simulation, Security**

I. Introduction

Vehicular systems have undergone rapid transformation in recent years, with the integration of autonomous features aimed at augmenting and, in many cases, replacing human operations. Autonomous features have the potential to dramatically increase safety by reducing human errors, while also allowing for more efficient use of transportation infrastructure and limiting environmental impacts. Nevertheless, one unintended consequence of this trend is an increase in electronic and software complexity. A typical vehicle on the road today includes hundreds of Electronic Control Units (ECU), tens to hundreds of sensors, an average of 3-5 in-vehicle networks, and hundreds of megabytes of software code. Obviously, these systems can contain subtle, hard-to-detect errors. A considerable number of these errors occur when multiple components are coordinated simultaneously. A simple optimization in one component, such as how a particular ECU interprets a message from the CAN bus, can have an unforeseen impact on timing, coordination, performance, or even functionality, which may jeopardize safety, reliability, and efficiency. Additionally, bugs that evade deployment can be exploited by adversaries in the field to introduce inefficiencies, and cause accidents, and finally, bugs that are discovered after they have been deployed can be fixed by the developers.

In current industrial practice, exploration of system functionality occurs through field testing at the sites of various automotive OEMs. This involves connecting all of the electronic parts, sensors, and actuators (possibly excluding the mechanical chassis), installing the necessary software, and exercising the vehicular use cases (*e.g.*, braking, traction control, turning right, etc.). A major drawback of this approach is that it can only be used after all the hardware, sensors, and actuators have been produced and put together. Any significant redesign is typically avoided at this point since it can cause a significant churn in the design and production timeline. This precludes potential opportunities for optimization that would be feasible if detected early in the design. Even design flaws found late can be expensive to address. Evidently, there is a critical need for a platform that can resolve all these issues while also allowing users to explore and validate system-level functionalities way early in the design process.

In this paper, we develop a novel prototyping environment, VIVE to address this critical problem. VIVE enables a user to systematically simulate a variety of automotive system-level use cases, understand how they interact, and develop system-level optimization and security solutions. To our knowledge, VIVE is the *first* prototyping environment for exploring system-level interactions in automotive systems. While there has been previous work on hybrid virtual platforms for individual hardware components [2], [3], they focus on the functionalities of a specific component rather than system-level interaction. On the other hand, there are automotive simulators (*e.g.*, SUMO [4], CARLA [5]). However, these systems enable the exploration of vehicular functionality in relation to their environment and the high-level algorithms involved in the realization of that functionality (see Section II-C). VIVE on the other hand focuses on the exploration of the system-level architecture of the vehicle, the coordination, and communication of ECUs through in-vehicle network messages, and their impacts on optimization and security. We are aware of no other platform that enables such exploration.

The paper makes the following important contributions:

- Our platform VIVE represents the first prototyping platform to enable early exploration and optimization of automotive system-level use cases.

This research has been supported in part by the National Science Foundation under Grant No. CNS-1908549 and SATC-2221900.

Md Rafiul Kabir, Bhagawat Baanav Yedla Ravi, and Sandip Ray are with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611, USA. Email: kabirm@ufl.edu, b.yedlaravi@ufl.edu, sandip@ece.ufl.edu.

Preliminary results from this research appeared in the Proceedings of the 24th IEEE International Conference on Intelligent Transportation [1].

- We discuss the configurability and extensibility needs of such a prototyping platform and demonstrate an approach to achieving this in VIVE.
- We demonstrate the use of VIVE in the design of a variety of representative vehicular use cases of varying complexity based on virtually simulated vehicular components.
- We demonstrate the use of VIVE in exploring optimization opportunities and critical security compromise scenarios.
- We showcase the integration and validation of VIVE with hardware setup in a real-time scenario.

The remainder of the paper is organized as follows. Section II discusses the background and relevant research in this area. Section III presents the system-level architecture of VIVE and the details of communications, components, and interfaces captured through the platform. In Section IV we discuss the use of VIVE for several vehicular use-case implementations. In Section V we demonstrate the application of the platform in system-level optimization and security solutions. Section VI presents the integration and validation of VIVE with the hardware setup. We conclude in Section VII. For the interested reader, a preliminary version of VIVE is available online [6].

II. Background and Related Work

A. Digital Twin

Virtual prototyping requires developing a software (*i.e.*, virtual) model of a design before constructing a physical embodiment. It allows any real-world system or component to be mimicked into a virtualized version that can be explored freely. A widely recognized prototyping method is known as *digital twins* [7]. The goal is to create a mathematical model that simulates the actual world in digital space using a digital representation of a physical device or item that mimics the underlying physics. This is typically done by combining models and data with state-of-the-art algorithms, expert design, and connectivity [7]. Applications for digital twins include many different fields *e.g.*, aerospace, manufacturing, medicine, and healthcare [8]. Furthermore, it has a few noteworthy applications in similar industries like aviation [9], [10]. However, the emphasis has been primarily on simulating mechanical and physical behavior.

B. Virtual Platform

The idea of virtual prototyping has been popular in the *cyber* world, targeting SoCs, embedded software, and digital hardware. This prototyping approach is well known as a *virtual platform*. It is a software-based modeling system that can completely mimic the functionality of a specific System-on-Chip (SoC) or digital hardware. The goal is to develop an abstract model of the underlying hardware device that can offer a configurable platform early in the design process (pre-silicon or even developed register-transfer level (RTL) models are ready), which can be utilized for software development and design optimization. However, this approach has been beneficial for early software and hardware development, verification & validation, and hardware-software co-design [11], [12]. Furthermore, virtual prototyping has been developed through a variety of commercial frameworks [13].

C. Related Work

As virtualization is becoming an emerging technology for the future of automotive safety and security, there has been some recent work to mention. Strobl *et al.* [2] discussed the utility of automotive virtualization in integrating several ECUs into a few Domain Controller Units (DCUs). They explained their approach with concepts like Virtual Computer System (VCS), Virtual Machine Monitor (VMM), etc. to suggest how to effectively map these concepts into the field of embedded systems —- especially those found in the automotive industry. In order to virtualize a GPU and the connected display device, Lee *et al.* [14] introduced the Virtualized Automotive DIsplay (VADI) system. VADI oversees two execution domains: one for the in-vehicle infotainment (IVI) software and one for the automotive control software. A VP was added to the V-model of automotive software development by Safar et al. [15] as part of an improved methodology that permits verification and validation at the ECU, SoC, and system level. Additionally, it offers a co-debugging mechanism and fault injection capability for AUTOSAR applications. Tharma *et al.* [3] discussed a digital twin method in the context of the automotive wiring harness in order to resolve the complexity of the wiring harness for passenger vehicles. It explained the basis of virtualization, specifically for wiring harness systems by using Digital-Mock-Up (DMU), virtual product development steps, and new concepts for the implementation of the Digital Twin method. Rajesh *et al*. [16] discussed the development of a digital twin that aids in the proactive maintenance of a car brake system. Using the ThingWorx Internet of Things (IoT) platform, brake pressure was recorded at various vehicle speeds as a proof of concept. While these techniques cover some components, unlike our framework, they only take certain subsystems into account. A Synthetic Data Vault (SDV) framework known as SOAFEE [17], was launched to provide a cloud-native architecture optimized for mixed critical automotive applications. It is an open architecture for the embedded edge that shows a high-level perspective of the hardware, software, and cloud components of an automotive central computing solution stack.

A variety of automotive simulators have been developed in recent years. SUMO [4] is an open-source traffic simulation framework that allows the simulation of traffic management strategies and vehicular communications. CARLA [5] is another open-source simulator for autonomous driving research that allows for the customization of sensor suites and environmental factors. Model-based designs using MATLAB/Simulink [18] for automotive systems have been used for design with simulation, implementation with code generation, continuous testing, and verification. AUTOSAR classic and adaptive platforms [19], [20] provide standardized software architecture for automotive ECUs. Each of these works focuses on a specific direction for automotive systems, *i.e.*, traffic simulation, autonomous driving environments, simulation for testing, cloud-native architecture, or software abstractions.

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3267339

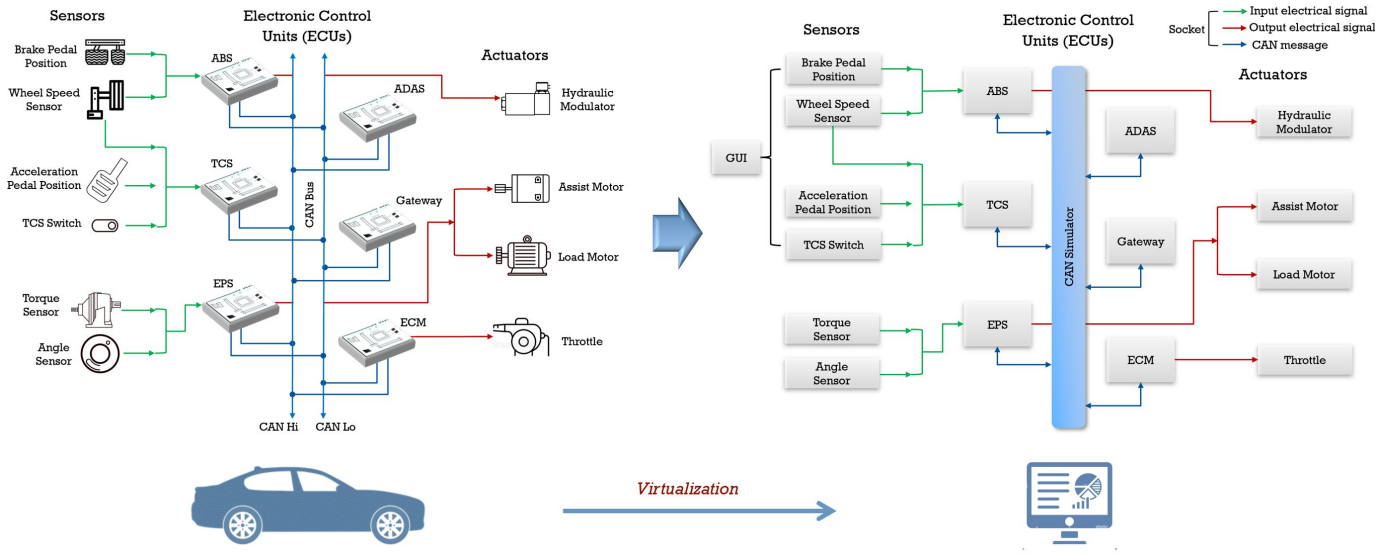

Figure 1. System architecture involving the virtualization of use cases. **(Left)** A typical physical prototype to support three representative use cases: Antilock Braking System, Traction Control System, and Right Turn. **(Right)** Corresponding Virtualization.

VIVE is different from (and complementary to) these prior efforts in its design goals, which governs the specific details of the system elaborated and abstracted in the platform. In particular, we focus on a prototyping solution that would enable a system architect to explore system-level interactions happening as a result of (possibly overlapping) vehicular use cases (*e.g.*, braking, turning, cruise control, security scenarios, etc.). The system-level interactions of interest for VIVE are interactions and communications among ECUs, sensors, and software. VIVE correspondingly provides a prototyping environment enabling exploration of configurability, optimization, and security arising from this interaction. Furthermore, VIVE differs from cloud-based SDF frameworks, being aligned with edge computing, hardware-software components, and system virtualization with inter-component and inter-system interactions. Finally, VIVE enables a wide range of integration possibilities through abstraction and configuration for both hardware and real software parameters.

III. VIVE ARCHITECTURE

VIVE derives inspiration from both digital twins and virtual platforms and aims to develop a targeted prototyping solution for automotive systems. Similar to digital twins, we target the total system-level interaction rather than a specific electronic component. Similar to virtual platforms, the emphasis here is on the cybernetic components of the system rather than its mechanical or physical characteristics. We focus on two features of cyber behavior:

1) Software functionality *i.e.*, involving indigenous computational processes.
2) Communication and coordination through electrical signals and CAN messages.

A. High-level Architecture

A modern automotive system consists of ECUs, perhaps connected with IoT components *i.e.*, sensors and actuators, as well as in-vehicle networks (*e.g.*, CAN, LIN). Every automotive use case (*e.g.*, brake, right turn, cruise, etc.) is initiated by some actuation action from the user and involves a series of messages transmitted over the network by various ECUs. Fig. 1 illustrates the high-level architecture of the platform indicating the virtualization involved in three illustrative use cases. Given this insight, VIVE permits the modeling of many use cases by providing the following infrastructure.

- **Network Simulator:** VIVE facilitates a simulation environment for the in-vehicle networks involved in the use case. The implemented protocols (*e.g.*, CAN, LIN, etc.) are modeled by the network simulator.
- **ECU:** Of course, each ECU is a sophisticated computational component. In contrast to conventional virtual platforms, a complete software implementation of the ECU is not necessary for VIVE. Instead, it offers a general interface that may be used to link either (1) an actual ECU, (2) a computation process of the functionality, or (3) a simple hardware platform (*e.g.*, Raspberry Pi) to mimic the computation of the ECU as required.
- **Sensors:** VIVE permits the interfacing with an actual sensor or just a software process to produce artificially generated computational data accurately simulating the behavior of automotive subsystem sensors (*e.g.*, angle sensor, wheel speed sensor, etc).
- **Actuators:** Analogous to sensors, the actuators are similarly represented by software processes and mostly function as the outputs from ECUs for actuation activities, (*e.g.*, assist motor, throttle, etc.). The actuator generates GUI blocks as needed that serve as the final output.

B. In-vehicle Communication

The communication system acts as the central hub for real-time simulation. VIVE models two types of communication: (1) Electrical signal, and (2) Communications via in-vehicle

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3267339

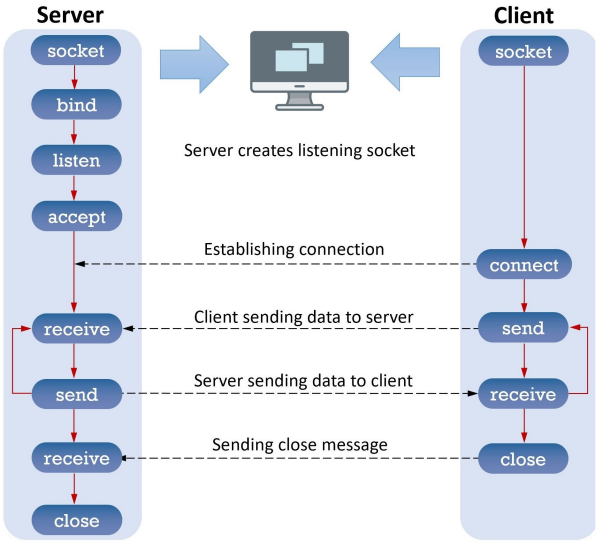

Figure 2. TCP client-server socket flow

networks. Our platform currently supports CAN communication. We developed a specific virtual module to prototype CAN bus communication *i.e.*, to be referred to as the CAN simulator. The simulator determines which ECU should it send data to by using a unique identifier *i.e.*, arbitration ID from the CAN frame. Apparently, each arbitration ID is associated with a port number that corresponds to a certain ECU. Furthermore, it can handle several messages from multiple ECUs across various use cases simultaneously.

Remark 1: (Implementation Note) The platform implementation for in-vehicle communication employs Transmission Control Protocol (TCP) [21] sockets to enable interaction among all the processes. TCP ensures that data packets are successfully transmitted across the network, though at a higher latency. Therefore, it can provide a secure and well-organized data transfer between several hosts via an IP network. The client-server model shown in Fig. 2 serves as the basis for our socket programming. Typically, the client initiates the communication while the server passively awaits to respond to the client's request [22]. Our implemented CAN simulator is modeled with a python-can library to build the CAN frame that is constructed as a byte array message for data transmission and reception via the socket. The message frame comprises the actual data with other peripheral information *i.e.*, arbitration id, extended id, and data length. The remaining sensor and actuator communication, *i.e.*, electrical signals (those that don't involve CAN frames), are simulated as simple byte-array messages. The CAN frame used in vehicles contains many components. However, for simulation, we chose a smaller frame structure from the CAN library.

C. Vehicular Component Models

Typically, an actuation activity by the user initiates the corresponding use case. For instance, traction Control will be initiated by a user pressing the acceleration pedal. Other components related to the use case (*e.g.*, ECU and sensor computations) execute constant, ongoing activity: the wheel speed sensor, while relevant to several use cases, performs ongoing activity continuously independent of the actuation actions that initiate the use case. VIVE supports this duality as follows: — Sensors (or processes simulating the sensory activities) are continuously sampled. Similarly, all computation blocks continue to run indefinitely. For instance, the implemented wheel speed sensor provides continuous vehicle speed information to the ECU. Subsequently, the acceleration pedal position sensor receives direct user input by applying a simulated acceleration pedal. (Fig. 4).

Note that the actuator occasionally uses information from the ECU to carry out specific mechanisms. As a result, the actuator computation process in VIVE supports the transmission of inputs from the simulated (or real) ECU and offers output data with the final outcome of ECU computation. For example, if we look at the Right Turn use case, the simulated assist motor receives input from the simulated Electric Power Steering (EPS) ECU and outputs what is required by the next mechanism as a result of the ECU computation. This output depicts the amount of steering assist torque being used for the right turn. Finally, the ECU process is linked with the (simulated) in-vehicle network in order to communicate with other ECUs. Unlike most sensors and actuators, the (simulated) ECU has the ability to both send and receive CAN messages. Furthermore, ECUs and sensors can participate in multiple use cases with necessary activities as required. For example, note from Table I, the participation of the ABS ECU and wheel speed sensor in four different use cases. We conclude the discussion on the sensor, actuator, and ECU models by noting the versatility of the platform.

Remark 2: (ECU Functionality) Our models of ECU computation for the use cases implemented are based on available open-source data (*e.g.*, the functionality of ABS [23]–[25]). Obviously, any deployed vehicle includes ECUs developed by suppliers, including confidential and proprietary design IPs. However, even in such cases, interface models are included in the design of the ECU to enable subsequent players in the supply chain to design and validate their interactions with the ECU. Such models provide sufficient collateral for incorporation into VIVE since VIVE focuses on the interaction of ECUs for realizing system-level use cases rather than low-level implementation details of the ECUs themselves. Indeed, this is one of the key features of VIVE that distinguishes it from field testing and enables early system-level validation before the ECU has been fully implemented. Once a (possibly proprietary) hardware implementation is ready, of course, VIVE also enables replacing the abstract interface model with the actual hardware to explore more elaborate details of the interaction.

D. Extensibility and User Interface

VIVE supports *extensibility*, *i.e.*, the seamless addition of new use cases (including those that might interact with the ones already simulated). In order to expand VIVE with a new use case, the following information needs to be provided:

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3267339

Table I
IMPLEMENTED USE CASES AND ASSOCIATED VEHICULAR COMPONENTS CLASSIFICATION

Name	ECU	Sensor	Actuator
Anti-lock Braking System (ABS)	ABS, ADAS, Gateway	Brake Pedal Position, Wheel Speed Sensor	Hydraulic Modulator
Right Turn	Electric Power Steering (EPS), ABS, Gateway	Angle Sensor, Torque Sensor	Assist Motor, Load Motor
Return-to-Center	ABS, EPS, Gateway	Angle Sensor, Torque Sensor	Assist Motor
Traction Control System (TCS)	TCS, ADAS, Gateway, Engine Control Module (ECM)	TCS Switch, Acceleration Pedal Position, Wheel Speed Sensor	Throttle
Cruise Control	Body Control Module (BCM), Gateway, Engine Control Module (ECM)	Cruise Switch, Acceleration Pedal Position, Wheel Speed Sensor, Acc/Dec Switch	Throttle
Indirect Tire Pressure Monitoring System (iTPMS)	ABS, Gateway	Wheel Speed Sensor	
Direct Tire Pressure Monitoring System (dTPMS)	Tire Pressure Monitoring System (TPMS), Gateway	Tire Pressure Sensor	
Ultrasonic Park Assist	Park Assist, EPS, BCM, Gateway	Ultrasonic Sensor	Gear Shifter

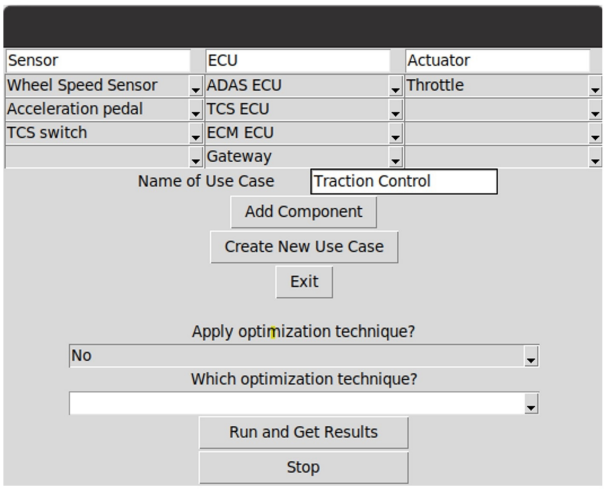

Figure 3. VIVE simulation window for use case initialization. (Shown for traction control here)

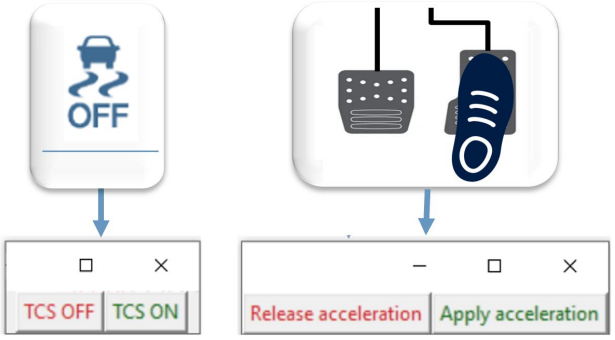

Figure 4. Traction Control GUI. Showing two user inputs: (1) TCS switch and (2) acceleration pedal position

1) ECUs involved; the required computation being performed by each ECU to actualize the functions of the ongoing use cases.
2) Sensors and actuators involved; when a new sensor or actuator is required for a use case, the process representing them must be included.
3) Communication involved; message sequences transmitted by the different ECUs from different use cases (and the modes of communication, *e.g.*, direct electrical signals, CAN, etc.).

The platform regulates resource sharing, scheduling, and communication between use cases. For example, recall the participation of wheel speed sensor in multiple use cases. Fig. 3 displays the simulation window from which a use case can be started (shown for traction control as an illustrative example). The window allows the choice of sensors, ECUs, and actuators to build the desired use case. It enables multiple graphical user interfaces (GUI) to perform actuation actions of pressing (shown in Fig. 4) and final simulation output.

Remark 3: (Real-time Computing) Since VIVE abstracts the implementation details of ECUs, interactions involved in use cases are explored in "event-driven" simulation at the level of CAN communications. One upshot of this design choice is that there is no direct synchronous global clock to account for real-time constraints. However, real-time constraints can be indirectly explored through congestion and latency measurements induced by the interaction of different use cases (see examples in Section V).

IV. USE CASES IMPLEMENTATION

Table I shows a list of use cases implemented in VIVE along with corresponding vehicular components. VIVE enables the user to define any arbitrary subset of the applicable use cases and examine their interactions during simulation. To give a sense of the platform operations, we provide a relatively detailed overview of two use case implementations; we then summarize the rest of the use cases.

A. Illustrative Use Case-1: ABS Implementation

Anti-lock Braking System (ABS) is a safety system to prevent the wheels from locking up during heavy braking, thus allowing the driver to maintain steer-ability control over the vehicle. Fig. 5 (a) provides an overview of the ABS use case. The functionality is fairly standard [23]–[25], which is simplified in Fig. 5 (b). The indigenous processes for

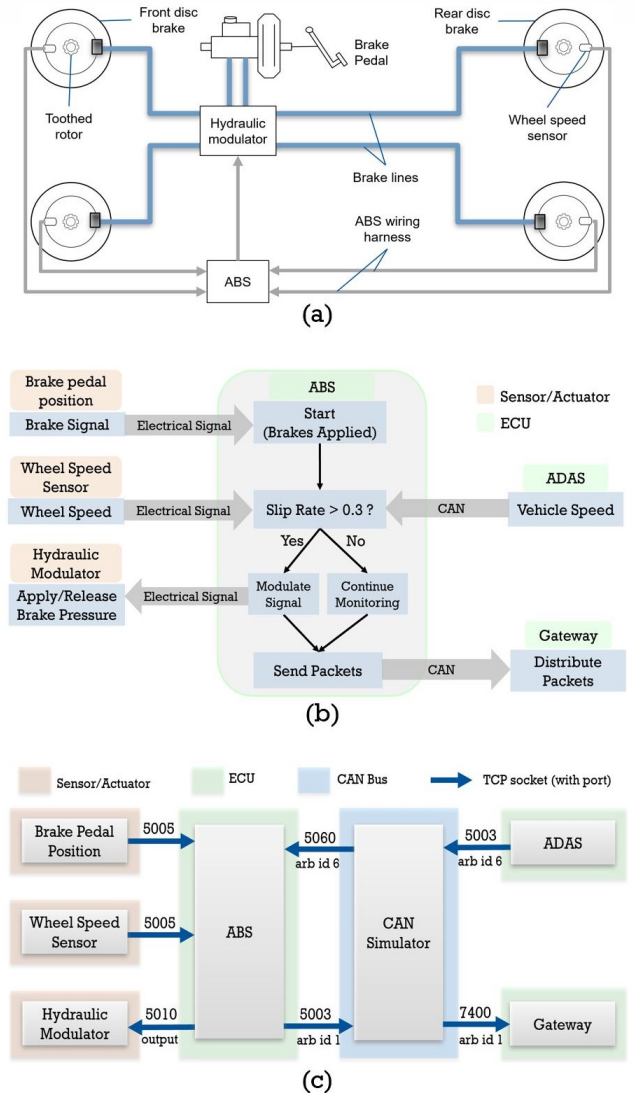

Figure 5. ABS use case **(a)** corresponding hardware architecture **(b)** functionality flow diagram and **(c)** primary system design with VIVE

vehicular components involved in this use case *i.e.*, ECUs (ABS, ADAS, and gateway), sensors (brake pedal position and wheel speed sensor), and actuator (hydraulic modulator) taking part in the primary system design are shown in Fig. 5 (c). The VIVE implementation utilizes "sockets" (*i.e.*, endpoint of a two-way communication link) for sharing data among these processes as discussed in Section III-B. For the ABS use case, the simulated brake pedal position, wheel speed sensor, and ADAS have client ports for sending data. The hydraulic modulator and the gateway have server ports for data reception. The ABS ECU and the CAN simulator are the only components with both client and server sockets with scheduling and computation capabilities based on the data. To establish a reliable connection, the client calls to connect to the server and initiates the three-way handshake. Each computation process is linked to a particular port number (*e.g.*, 5005, 7400, etc) and additionally each ECU is recognized by arbitration IDs (*e.g.*, 1 and 6) *i.e.*, shown in Fig. 5 (c). The port numbers between communicating sockets (*i.e.*, simulation processes) have to match to establish a secured connection.

Table II
BYTE ARRAY MESSAGES FOR ABS

Component	Message
Brake Pedal Position	[0] or [1]
Wheel Speed Sensor	simulated values ranging between [0] - [60]
ADAS	simulated values ranging between [0] - [60] (CAN)
ABS (final output)	[0] or [1000] (CAN)
ABS (for gateway)	[0] or [1] (CAN)

The user initiates the ABS use case by a certain actuation action *i.e.*, applying the brake, which is enabled via the VIVE GUI. Table II shows all the byte array messages (via socket) by each component. On the action, the system behaves as follows.

- The ABS ECU receives the brake position signal from the brake pedal position sensor after the user input.
- The slip rate is computed using the simulated speed data from the wheel speed sensor and ADAS. For simulation purposes, the speed values range between 0-60 mph. However, this range is not intuitively limited to every use case.
- If the slip rate exceeds 0.3, the ABS notifies the hydraulic modulator through a byte array message to either apply or release brake pressure.
- Since the brake pressure parameters in a vehicle vary between 1000 psi and 1600 psi [26], the byte array message sent to the hydraulic modulator is depicted as `[0]` and `[1000]` for no pressure and pressure application, respectively.
- Due to the fact that this process runs continuously, the slip rate fluctuates based on various speed data, producing two different final outputs (ABS active or not active).
- Additionally, the ABS ECU sends a CAN message (containing the ongoing ABS activity) to the CAN simulator with an arbitration ID of 1, designating the gateway ECU.
- Finally, the gateway routes the ABS packets to other ECUs and the instrument cluster via another CAN bus.[1]

Note that, while running ABS, any other use case can take place *i.e.*, multiple ECUs can communicate with the CAN simulator at the same time, which does not hamper any of the ABS functionalities. Similarly, among ABS use case components, the CAN simulator handles both the ABS and ADAS ECUs with queuing and coordination capabilities, allowing them to securely communicate and not lose any data. This enables the user to comprehend the effects of several interactions and the need for any substantial re-design, *e.g.*, to support bandwidth or real-time constraints of the target vehicle.

B. Illustrative Use Case-2: UPA Implementation

Ultrasonic Park Assist (UPA) system assists the driver in parking and avoiding objects while in reverse gear. It operates

[1]VIVE component models include the gateway but not the instrument clusters. Consequently, any response from the instrument cluster is addressed directly from the gateway.

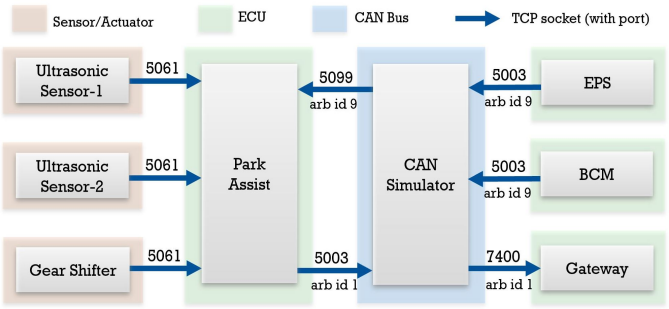

Figure 6. UPA use case primary system design with VIVE

at a very low speed *i.e.*, 5 mph. The functionality is reasonably standard [27], [28] and it works as an alert/warning-based system. The system primarily works for two types of parking scenarios: 1) perpendicular and 2) parallel parking. The indigenous processes for vehicular components involved in this use case are: UPA, BCM, EPS, and gateway ECUs, two ultrasonic sensors,[2] and a gear shifter. The primary system design (including all the operating port numbers and arbitration IDs) is shown in Fig. 6. Note that the system itself does not require observing the functions of components like EPS or brake pedal, but rather our use case has those components to show the maneuvering status of the driver. On the action, the system behaves as follows.

- After applying the reverse gear, the ultrasonic sensors get activated to sense objects.
- The driver's maneuver status positions are known from the CAN message containing the steering angle and brake position information provided by the EPS and BCM ECUs, respectively (via arbitration ID 9).
- For this initial implementation, object presence (*e.g.*, parked car) is based on user input to the ultrasonic sensor.
- At the point of collision, the sensors will send an electrical signal to the park assist ECU as a byte array message of either `[0]` or `[1]`.
- Next, the ABS ECU sends a CAN message (containing the ongoing UPA activity) to the CAN simulator with an arbitration ID of 1, designating the gateway ECU.
- Finally, the gateway is supposed to route the packets to other ECUs and the instrument cluster. The dashboard warnings are depicted as GUI output from the gateway itself.

C. Other Use Cases Summary

VIVE incorporates various automotive subsystems to enable its flexibility and viability. All the use cases involve multiple ECUs, sensors, and actuators, with some shared across the use cases. It was challenging to implement the right functionalities for all use cases, as the same subsystem functions differently in different vehicles in the real world. We attempted to prioritize use cases that have noticeable functionality that is shared by the majority of automobile manufacturers. We provide a brief discussion of all the remaining use cases below.

[2]The number of ultrasonic sensors varies in different implementations. In our implementation, we use two sensors.

1) Right Turn: The right turn use case is a segment of the electric power steering system (EPS) that helps drivers steer the vehicle with minimal effort needed to turn the steering wheel. The functionality follows a standard control logic [29]. The torques are categorized as load torque, assist torque, and steering torque. The data provided by the angle sensor and torque sensor on the steering wheel simulate the user turning to the right. The EPS ECU calculates the assist and load torques required. Consequently, ABS transmits the vehicle speed as a CAN message to EPS, and EPS responds by transmitting the turn status to the gateway.

2) Return-to-Center (RTC): The RTC use case is another segment of the electric power steering system that occurs after making a right or left turn by steering the wheels. The angle sensor and torque sensor simulate the steering wheel returning to the center after executing the right turn. The required RTC assist torque is calculated by the EPS ECU. In response, EPS communicates the RTC status as a CAN frame to the gateway after receiving the vehicle speed as a CAN frame from ABS.

3) Traction Control System (TCS): Our implemented traction control use case follows a basic functionality [30], [31], where the objective is to change the torque during vehicle acceleration based on the calculated slip rate. The VIVE GUI allows the user to launch actuation operations by pressing the TCS switch and adjusting the acceleration pedal position. TCS is activated while accelerating. Similar to the ABS use case, slip rate calculation from simulated speed values induces further actions *i.e.*, ADAS sends the vehicle speed as a CAN frame to TCS, and in response, the torque reduction status is sent as a CAN message from TCS to both the ECM and the gateway. The final output from the ECM depicts the transmission of torque reduction data to the throttle for providing traction, allowing the vehicle to accelerate more smoothly.

4) Cruise Control: The cruise control use case has multiple GUIs to potentially represent the acceleration pedal, cruise ON/OFF switch, and cruise acceleration/deceleration buttons. The user has to activate the cruise control to increase, decrease, or maintain the vehicle speed. The functionality is kept as minimal as possible for basic interactions. The ECM ECU takes sensor inputs *i.e.*, inputs from the GUI via BCM as a CAN message, and computes the speed direction. Based on the computation, the ECM sends the cruise status as a CAN message to the gateway. This whole sequence happens in one cycle. As the cruise control system is a continuously running process, there is feedback implemented that goes back to the GUI inputs for cruise status and current speed values. Consequently, the next cycles run, taking the feedback as initial values.

5) Indirect Tire Pressure Monitoring System (iTPMS): This use case works with the car's ABS and wheel speed sensors (for systems that do not have tire pressure sensors inside the tires). If one of the tires' pressures is low, it will roll at a different wheel speed than the others. The ABS ECU detects this information by calculating tire pressure from the wheel speed sensors to check pressure status. Subsequently, a tire

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3267339

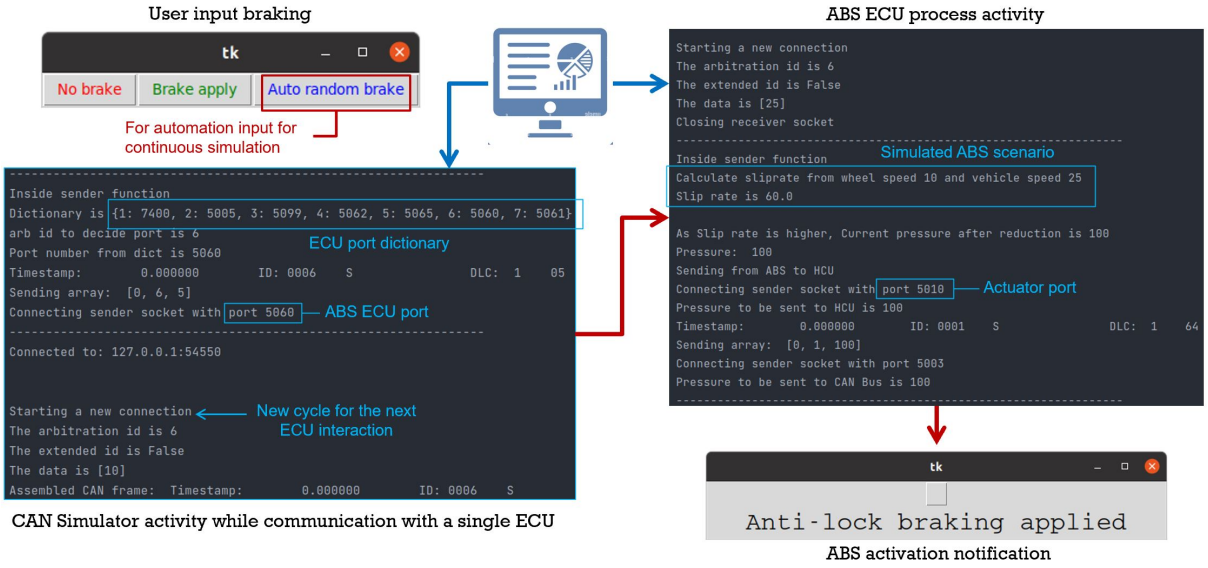

Figure 7. ABS use case front-end activity

pressure alert message is sent as a CAN frame from ABS to the gateway.

6) Direct Tire Pressure Monitoring System (dTPMS): This use case does not depend on ABS; rather, it has its own ECU and sensors to monitor the tire pressure. The central ECU responsible for the Tire Pressure Monitoring System (TPMS), directly receives continuous tire pressure readings from pressure sensors. Failing to maintain the minimum pressure mark will trigger the transmission of the tire pressure warning data as a CAN message from the TPMS to the gateway.

This allows for the application of more complicated integrations at later phases of system development when actual hardware and software components are present. Note that the RPI integration can be effective while simulating not a few use cases, but rather a large number (*e.g.*, 50-100) of use cases where significantly more computation has to take place together, especially with the CAN simulator handling multiple ECUs with adequate efficiency and optimization.

D. Front-end Activity and Real-world Data

ViVE provides an elaborate front-end for the user to initiate realistic use cases, comprehend interactions among various sub-systems and components, and estimate various kinds of latency overheads. Based on various customizations of use cases, a realistic simulation is performed with CAN simulator performance, ECU computation results, and sensor/actuator activities. After initiating a use case from the simulation window, we observe all the activities for each simulation process within the platform window. Fig. 7 shows the experimental front-end activity for the ABS use case, illustrating the activities of CAN simulation and the ABS ECU process here, as the major computations take place in these two components. It shows how the CAN simulator is detecting corresponding ECUs and the related computational activities of the ABS ECU. Moreover, the platform allows exploration of simulated and actual real-world data from sensors like wheel speed sensor and comprehends how the simulated speed data is perceived by the respective ECUs. Section VI explores this feature of ViVE to connect to perform integration validation with physical models.

Remark 4: (Technical challenges) The CAN protocol is designed to achieve a number of potentially disparate goals. Consequently, the CAN payload includes a significant amount of information, which can be confusing to the user. So, ViVE carefully extracts a specific subset of the CAN frame that is useful for the exploration of specific use cases. For instance, instead of using a full CAN frame, we used a smaller, simpler frame that would be more informative to the user. Mimicking the CAN networking with socket programming was done to accumulate all the functionalities of the CAN hardware into a single process. Another challenge was to mimic the right functionalities for all use cases, as the same subsystem functions differently in different vehicles in the real world. We prioritize use cases that have noticeable functionality that is shared by the majority of automobile manufacturers.

V. Applications beyond Exploration: Optimization and Security with ViVE

Since ViVE enables exploration of various scenarios and their interactions, users have the opportunity to comprehend bottlenecks on various parameters and "tweak" the design to address these bottlenecks early in the system development. Correspondingly, it enables early exploration of *rare* interactions to analyze potential safety and security impacts. In this section, we showcase these abilities by developing applications on top of ViVE to address specific optimization and security targets. Note that the *conclusion* from these applications are not germane to the paper, *e.g.*, the bottlenecks discovered by the optimizer depend on the specifics of the use case implementations being used and would vary if the implementations were modified. However, our work shows how the same approach can be used by OEMs to analyze

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3267339

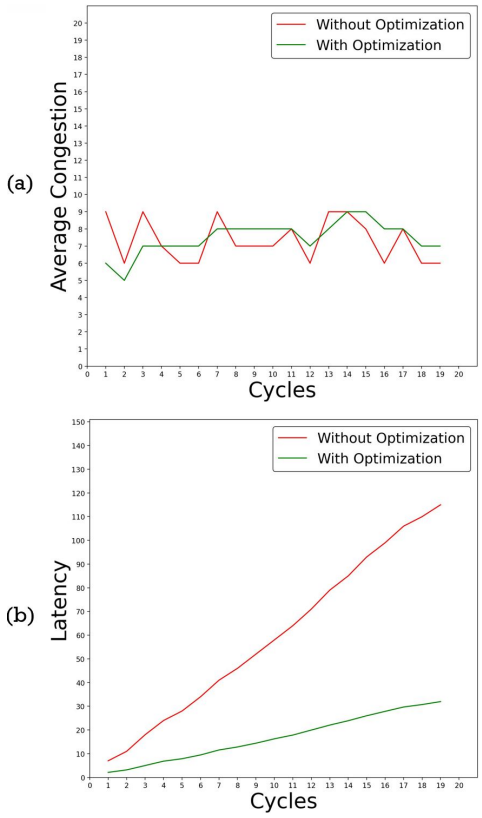

Figure 8. Results of optimization with Simulated Annealing on VIVE. The illustrated use cases for this scheduling optimization are ABS, Right Turn, Return-to-Center, and TCS. **(a) Congestion vs. Time. (b) Latency vs. Time.**

optimization and security concerns for their vehicular use cases by porting their use case models with VIVE.

A. Optimization

To demonstrate optimization, VIVE includes the implementation of two (independent) real-time CAN packet scheduling optimizers:

1) to reduce CAN bandwidth, *i.e.*, the number of CAN packets sent simultaneously.
2) to minimize latency, *i.e.*, the amount of time it typically takes for a CAN packet to travel from its source ECU to its destination ECU.

We employ Simulated Annealing [32] for our real-time optimizer.[3] By conceptualizing the idea of *slow cooling* of metals as a gradual drop in the likelihood of accepting inferior solutions as the feasible region is investigated, it provides a close approximation to the optimal solution. In the real-time implementation, the CAN bus transmits the packets specified in the first cycle of the sequence once the final temperature is achieved and the estimated sequence is determined, while delaying the transmission of the remaining packets until the bus cycle after that. The optimization values obviously depend on the use cases taken into account, and the absolute numbers

[3]Obviously, one can implement a superior optimizer on top of the user cases as well. Since the goal of the paper is to show the *feasiblity* of implementing optimizers rather than their sophistication, we choose a baseline optimizer for this demonstration because of its ease of use and comprehensibility.

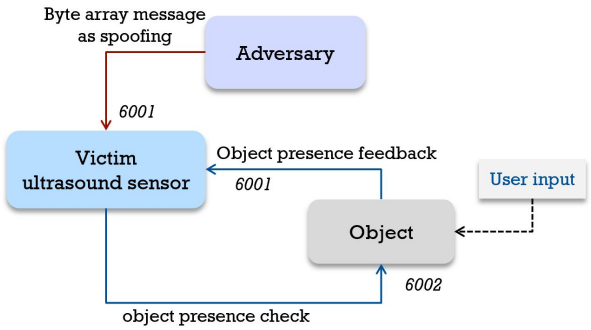

Figure 9. Ultrasonic sensor security compromise exploration with VIVE

might change for different implementations and use case combinations. The plots, however, highlight the platform's usefulness for real-time optimizations in practical use. Note that CAN packet scheduling to optimize for latency has a considerable impact on the results for the use cases investigated, whereas optimization for congestion appears to have a minimal effect.

Remark 5: (CAN Scheduling Note) The reason why optimization for congestion has such an outcome and even performs worse at some cycles than non-optimized scheduling is that the number of packets delayed from cycle t to cycle $t+1$ in order to preserve bandwidth at cycle t exceeds the number of new packets that the bus receives at cycle $t+1$. Such insights obviously allow for systematic and efficient CAN scheduling if they are attained early in the design exploration process.

Note that since system-level instances like the ones addressed in this research are only tested late during field testing, optimizing packet scheduling must be done offline without taking into consideration how various use cases interact. However, with VIVE, the user can explore the results of this interaction early in the design and fine-tune the optimizer accordingly.

B. Security

Modern automobiles with autonomous features have the potential to significantly increase safety by reducing and eventually eliminating human error. However, an obvious result of autonomy is that these systems are now more vulnerable to cyberattacks. Unfortunately, despite its utmost importance, public awareness on the issue of security in vehicular systems is still very low. Major security compromises on ultrasonic sensors have been investigated in automotive security research, based on spoofing and jamming attacks. Although the precise mechanisms vary depending on the attack scenario, some concepts are applied in various attacks. In particular, a *spoofing attack* is a well-known adversarial method for compromising ultrasonic sensors that involves the emission of ultrasonic pulses that are analogous to the victim sensor [33]. It entails modifying the return signal so that the victim vehicle receives a false signal before the actual echo. Consequently, the sensor in the victim vehicle interprets the spoofing signals in the same way as the actual signals, leading to false obstacle detection [34].

VIVE enables exploration of attack scenarios for ranging sensor attacks. We showcase this ability with the Park Assist use case. Fig. 9 shows the workflow for a (simplified) spoofing attack scenario compromising the ultrasonic sensor. The ultrasound signal is represented as byte-array messages sent to the object, which is a user-based model for object detection. Three computation processes represent the ultrasonic sensor (as the victim), the adversary sensor (attacker), and the object to be detected (*e.g.*, parked car, wall, stop sign, etc.). On the action, the victim ultrasonic sensor sends a byte array message to the object (via port 6002) to calculate its presence. The user input GUI provides the status, and consequently, the object computation module informs its status (`[0]` for absence and `[1]` for presence) back to the ultrasound sensor. The adversary can attack the sensor by sending similar (to mimic spoofing) byte-array messages. As the ultrasonic sensor receives all messages using TCP port 6001, a spoofing attack (a false echo from the adversary) leads to the false detection of the object *i.e.*, the vehicle assuming the presence of an object that is either closer or farther away than the actual distance.

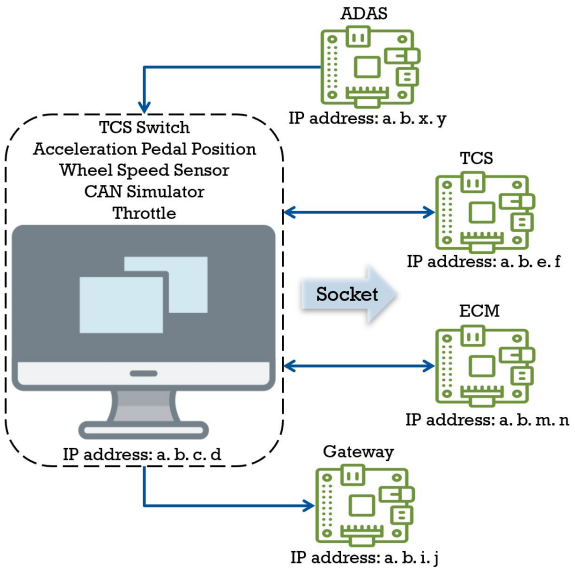

Figure 10. Raspberry Pi integration for Traction Control; showing one personal computer and four Raspberry Pis containing all components

VI. Integration and Validation

With all the implemented use cases and the applications beyond exploration, we establish VIVE as primarily adhering to the indigenous process models since we put a lot of emphasis on testing system-level scenarios early in the design process. Nevertheless, in VIVE, switching out a component for a more complex one only requires a simple component switch; the interfaces themselves are left unchanged. In this section, we showcase two specific applications of this extensibility. First, we discuss an integration with Raspberry Pi that enables early validation of automotive software. Second, we present integration with physical sensors and actuators in a vehicle model, which enables us to validate the platform's computation model, on which the use cases have been implemented, and the viability of the infrastructure as a backbone for use cases with real-time requirements.

A. Raspberry Pi Integration

To facilitate the exploration of functionality using real software, all use cases can also be realized with Raspberry Pi models for the corresponding ECUs. In Fig. 10, we show the integration of the Raspberry Pi with Traction Control as an illustrative use case. Here, all the computational blocks for the corresponding sensors, actuator, and CAN bus are implemented individually, while the ECUs (*i.e.*, ADAS, TCS, ECM, and gateway) are realized through RPIs. All five devices will have their own IP addresses to connect to for this integration, resulting in the development of an embedded system with more computational capability. Because the RPI can integrate with actual physical sensors, VIVE provides an interface to integrate with physical sensors rather than computational processes.

B. Configurability and Validation using PiCar

We now showcase the integration capability of VIVE with physically configurable automotive electronics for validating the viability of the infrastructure as a means of prototyping use cases with real-time constraints. The vehicle we employ for the test case detailed in this article is a PiCar-X by Sunfounder, which is an open-source robot learning platform built on the Raspberry Pi [35].

Platform Integration: Initiating the integration, all the processes of VIVE environment are run on a single Raspberry Pi mounted on the Pi-Car. Recall from the ABS illustration, the speed values and the slip rate are critical for the use case activity. While VIVE ABS use case comprises the ADAS ECU process with simulated speed values, we employ the ultrasonic sensor of the PiCar to get real-time vehicle speed data. The actuarial function from the brake pedal position of VIVE is integrated with the PiCar controller module. The PiCar consists of two DC motors that propel it and a servo motor that steers it. Due to the gear reduction unit in the propulsion system, the pi car is incapable of coasting due to kinetic inertia. Hence, this locks up the wheels. We intend to utilize this outcome as a hard braking condition. This allows us to utilize the car without having to include some additional form of braking system.

Environmental Design: The Pi car is incapable of moving at speeds high enough to recreate a skidding condition due to its low top speed and low inertia. To understand and validate our ABS, an inclined surface is set up where the car is allowed to drive downward. Upon disabling the power to the propulsion system, the wheels lock up because of the gear reduction mechanism mentioned previously. This lets the Pi car slide down the incline with the wheels locked. The angle of inclination is decided after multiple trials to recreate the condition that is easiest to perceive. Fig. 11 shows the hardware exploitation scenario.

Experiment and Evaluation: The PiCar uses a PWM signal as input from the VIVE environment to control the speed of the propulsion system. This feature can transition into real-

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3267339

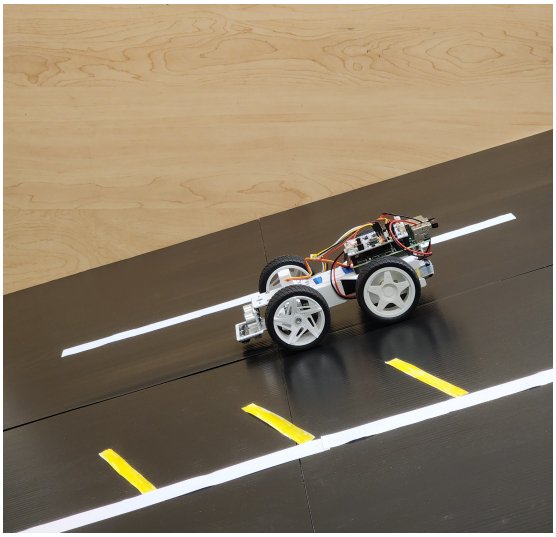

Figure 11. PiCar exploration testbed

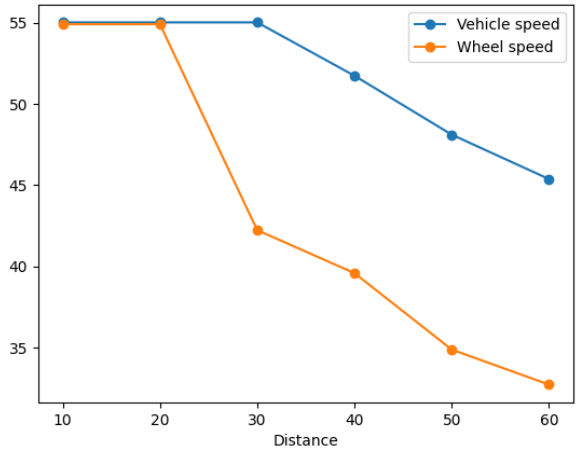

Figure 12. VIVE enabling vehicle speed and wheel speed changes with distance in the PiCar

time speed, assuming that there is very little backlash. This was validated using a setup that involved a flat, zero-inclined road to evaluate speed with respect to PWM input. This provided a correlation that was uniform. Now that the wheel speed values are obtained, we need to calculate the actual speed of the vehicle, including the condition of slipping. For this, we gathered the ultrasonic sensor data from the onboard ultrasonic sensor. A zero incline surface would produce the same result as the ultrasonic sensor-based speed value because there is no slip in flat surface conditions. However, for the inclined surface condition due to locking and sliding actual vehicle speed is higher than the wheel speed. We have also measured the distance values for various initial speeds just before brakes were applied to compare the braking distances and, subsequently, to compare ABS inactive vs. ABS active conditions. At the point of 20 cm distance (Fig. 12), the ABS ECU process running in the PiCar receives the brake position signal from the user input that enables the wheel speed value to start deviating from vehicle speed, resulting in an appropriate slip condition *i.e.*, slip-rate exceeding 0.3. Therefore, we observe the ABS activation as we gradually move towards a lower speed (simulating the vehicle slowing down). We further evaluated the trend of wheel speed and vehicle speed relation (from Fig. 12) with existing research [36], [37]. We conclude that VIVE can be reconfigured to be integrated with a hardware setup for real-time simulation exploration and can be a viable option for shortening the sim-to-real gap.

Remark 6: (Real-World Testing) As the integration validation demonstrates, VIVE supports hardware-in-loop testing by enabling the connection of various sensors and actuators, *e.g.*, the ABS use case mentioned above. This is possible due to the versatile nature of the hardware integration with the onboard Raspberry Pi. To simulate the functioning of an ECU, a microcomputer such as a Raspberry Pi Zero or a microcontroller such as a Raspberry Pi Pico can be used. The peripherals like sensors and actuators can be interfaced with these ECUs to perform a task and communicate with other ECUs using serial communication. As Section VI-B demonstrates, simulation achieves results very close to real-world testing.

VII. Conclusion

We have discussed a prototyping solution VIVE for methodically testing vehicular use cases at the system level. VIVE is a versatile, configurable platform that allows use cases to be specified using components at various abstraction levels, *e.g.*, indigenous processes, real hardware, or a Raspberry Pi. Furthermore, the platform supports extensibility with new use cases. Using VIVE, we demonstrated a number of use cases in practice and showed how it is useful for developing real-time scheduling optimizers. Additionally, VIVE enables versatile exploration for various security compromises. Finally, our platform demonstrates an effective roadmap for early system-level exploration of diverse cyber-physical subsystems.

In future work, we plan to extend VIVE with more critical use cases and employ other optimization algorithms. We will also explore the utility of this platform to conduct some preliminary research and early validation on functional safety and security properties with adversarial attack scenarios.

References

[1] M. R. Kabir, N. Mishra, and S. Ray, "Vive: Virtualization of vehicular electronics for system-level exploration," in *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2021, pp. 3307–3312.

[2] M. Strobl, M. Kucera, A. Foeldi, T. Waas, N. Balbierer, and C. Hilbert, "Towards automotive virtualization," in *2013 International Conference on Applied Electronics*. IEEE, 2013, pp. 1–6.

[3] R. Tharma, R. Winter, M. 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.

[4] M. Behrisch, L. Bieker, J. Erdmann, and D. Krajzewicz, "Sumo–simulation of urban mobility: an overview," in *Proceedings of SIMUL 2011, The Third International Conference on Advances in System Simulation*. ThinkMind, 2011.

[5] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," in *Conference on robot learning*. PMLR, 2017, pp. 1–16.

[6] M. R. Kabir, B. B. Yedla Ravi, and S. Ray, "Virtualization of Vehicular Electronics," http://sandip.ece.ufl.edu/projects/vive/main.html.

[7] D. Wagg, K. Worden, R. Barthorpe, and P. Gardner, "Digital twins: State-of-the-art and future directions for modeling and simulation in engineering dynamics applications," *ASCE-ASME J Risk and Uncert in Engrg Sys Part B Mech Engrg*, vol. 6, no. 3, 2020.

[8] B. R. Barricelli, E. Casiraghi, and D. Fogli, "A survey on digital twin: Definitions, characteristics, applications, and design implications," *IEEE access*, vol. 7, pp. 167 653–167 671, 2019.

[9] C. Li, S. Mahadevan, Y. Ling, S. Choze, and L. Wang, "Dynamic bayesian network for aircraft wing health monitoring digital twin," *Aiaa Journal*, vol. 55, no. 3, pp. 930–941, 2017.

[10] A. J. Zakrajsek and S. Mall, "The development and use of a digital twin model for tire touchdown health monitoring," in *58th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2017, p. 0863.

[11] S. Ahn and S. Malik, "Automated firmware testing using firmware-hardware interaction patterns," in *CODES+ISSS*, 2014, pp. 25:1–25:10.

[12] R. Kannavara, , C. J. Havlicek, B. Chen, M. R. Tuttle, K. Cong, S. Ray, and F. Xie, "Challenges and Opportunities in Concolic Testing," in *Proceedings of the National Aerospace Electronics Conference – Ohio Innovation Summit*, 2015.

[13] Synopsys, "Virtualizer," https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html, 2019.

[14] C. Lee, S.-W. Kim, and C. Yoo, "Vadi: Gpu virtualization for an automotive platform," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 1, pp. 277–290, 2015.

[15] M. Safar, M. A. El-Moursy, M. Abdelsalam, A. Bakr, K. Khalil, and A. Salem, "Virtual verification and validation of automotive system," *Journal of Circuits, Systems and Computers*, vol. 28, no. 04, p. 1950071, 2019.

[16] P. Rajesh, N. Manikandan, C. Ramshankar, T. Vishwanathan, and C. Sathishkumar, "Digital twin of an automotive brake pad for predictive maintenance," *Procedia Computer Science*, vol. 165, pp. 18–24, 2019.

[17] "Soafee homepage," 2022. [Online]. Available: https://www.soafee.io/

[18] J. Friedman, "Matlab/simulink for automotive systems design," in *Proceedings of the Design Automation & Test in Europe Conference*, vol. 1. IEEE, 2006, pp. 1–2.

[19] M. Staron and M. Staron, "Autosar (automotive open system architecture)," *Automotive Software Architectures: An Introduction*, pp. 97–136, 2021.

[20] S. Fürst and M. Bechter, "Autosar for connected and autonomous vehicles: The autosar adaptive platform," in *2016 46th annual IEEE/IFIP international conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, 2016, pp. 215–217.

[21] J. Postel, "Transmission control protocol," Tech. Rep., 1981.

[22] R. L. R. Maata, R. Cordova, B. Sudramurthy, and A. Halibas, "Design and implementation of client-server based application using socket programming in a distributed computing environment," in *2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*. IEEE, 2017, pp. 1–4.

[23] V. D. Gowda, A. Ramachandra, M. Thippeswamy, C. Pandurangappa, and P. R. Naidu, "Modelling and performance evaluation of anti-lock braking system," *J. Eng. Sci. Technol*, vol. 14, no. 5, pp. 3028–3045, 2019.

[24] T. Matsushita, K. Kondo, T. Yasuda, and H. Watanabe, "Abs control unit," *Fujitsu Ten Tech*, vol. 6, pp. 52–62, 1994.

[25] A. A. Aly, E.-S. Zeidan, A. Hamed, F. Salem, *et al.*, "An antilock-braking systems (abs) control: A technical review," *Intelligent control and Automation*, vol. 2, no. 03, p. 186, 2011.

[26] ASE, "Abs accumulators," 2008. [Online]. Available: https://www.freeasestudyguides.com/abs-pump-accumulator.html

[27] G. P. Zobel, "Warning tone selection for a reverse parking aid system," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 42, no. 17. SAGE Publications Sage CA: Los Angeles, CA, 1998, pp. 1242–1246.

[28] GMC, "Ultrasonic parking assist," 2022. [Online]. Available: https://www.gmcmaster.com/gimes-327-ultrasonic-parking-assist.html

[29] J.-H. Kim and J.-B. Song, "Control logic for an electric power steering system using assist motor," *Mechatronics*, vol. 12, no. 3, pp. 447–459, 2002.

[30] J. H. Park and C. Y. Kim, "Wheel slip control in traction control system for vehicle stability," *Vehicle system dynamics*, vol. 31, no. 4, pp. 263–278, 1999.

[31] L. Austin and D. Morrey, "Recent advances in antilock braking systems and traction control systems," *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, vol. 214, no. 6, pp. 625–638, 2000.

[32] K. Dharageshwari and C. Nayanatara, "Multiobjective optimal placement of multiple distributed generations in ieee 33 bus radial system using simulated annealing," in *2015 International Conference on Circuits, Power and Computing Technologies [ICCPCT-2015]*, 2015, pp. 1–7.

[33] W. Xu, C. Yan, W. Jia, X. Ji, and J. Liu, "Analyzing and enhancing the security of ultrasonic sensors for autonomous vehicles," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 5015–5029, 2018.

[34] B. B. Y. Ravi, M. R. Kabir, N. Mishra, S. Boddupalli, and S. Ray, "Autohal: An exploration platform for ranging sensor attacks on automotive systems," in *2022 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 2022, pp. 1–2.

[35] "Sunfounder picar-x: Self-driving robot car for the raspberry pi platform." [Online]. Available: https://docs.sunfounder.com/projects/picar-x/en/latest/

[36] D. V. Gowda, R. AC, M. Thippeswamy, C. Pandurangappa, *et al.*, "Automotive braking system simulations v diagram approach," *International Journal of Engineering & Technology*, vol. 7, no. 3, pp. 1740–1744, 2018.

[37] M. Watany *et al.*, "Performance of a road vehicle with hydraulic brake systems using slip control strategy," *American Journal of Vehicle Design*, vol. 2, no. 1, pp. 7–18, 2014.

Md Rafiul Kabir is a Ph.D. student at the Department of Electrical and Computer Engineering, University of Florida. Before that, he was an Electrical Engineer at Horizon Global Americas in Michigan, where he worked in the design and development of OEM Trailer Brake Controllers used in automobiles. Prior to that, he got his MSc degree in Electrical Engineering from the University of Toledo, where his research was on renewable energy applications. He received his B.Sc. degree in Electrical and Electronic Engineering from Ahsanullah University of Science and Technology, Dhaka, Bangladesh. Rafiul's current research interests are digital twins, exploration platforms for vehicular systems, cybersecurity and IoT applications.

Bhagawat Baanav Yedla Ravi is currently pursuing his Ph.D. in Electrical and Computer Engineering. He received his Master's degree in Mechanical Engineering from the University of Florida. He has 2 years of end-to-end product design and development experience with consumer electronics and automotive systems. His research interests include vehicular security, exploration, and learning platforms across IoT and vehicular systems.

Sandip Ray (SM'13) is a Professor at the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, where he holds an Endowed IoT Term Professorship. Before joining the University of Florida, he was a Senior Principal Engineer at NXP Semiconductors, and prior to that, Research Scientist with the Intel Strategic CAD Laboratories. Dr. Ray's current research targets correct, dependable, secure, and trustworthy computing through the cooperation of specification, synthesis, architecture, and validation technologies. He is the author of three books and over 100 publications in international journals and conferences. He has also served as a Technical Program Committee Member of over 50 international conferences, as Program Chair of ACL2 2009, FMCAD 2013, and IFIP IoT 2019, as Guest Editor for IEEE Design and Test, IEEE TMSCS, and ACM TODAES, and as Associate Editor of Springer HaSS and IEEE TMSCS. Dr. Ray has a Ph.D. from the University of Texas at Austin.