A Distributed Safety Mechanism using Middleware and Hypervisors for AutonomousVehicles

Tjerk Bijlsma
Embedded Systems Innovations
TNO
Eindhoven, the Netherlands
tjerk.bijlsma@tno.nl

Andrii Buriachevskyi
CTO
NXP Semiconductors
Eindhoven, the Netherlands
andrii.buriachevskyi@nxp.com

Alessandro Frigerio
Electrical Engineering
Eindhoven University of Technology
Eindhoven, the Netherlands
a.frigerio@tue.nl

Yuting Fu
CTO
NXP Semiconductors
Eindhoven, the Netherlands
yuting.fu_1@nxp.com

Kees Goossens
Electrical Engineering
Eindhoven University of Technology
Eindhoven, the Netherlands
k.g.w.goossens@tue.nl

Ali Osman Örs
BL AMP
NXP Semiconductors
Ottawa, Canada
ali.ors@nxp.com

Pieter J. van der Perk
CTO
NXP Semiconductors
Eindhoven, the Netherlands
peter.vanderperk@nxp.com

Andrei Terechko
CTO
NXP Semiconductors
Eindhoven, the Netherlands
andrei.terechko@nxp.com

Bart Vermeulen
CTO
NXP Semiconductors
Eindhoven, the Netherlands
bart.vermeulen@nxp.com

***Abstract*—Autonomous vehicles use cyber-physical systems to provide comfort and safety to passengers. Design of safety mechanisms for such systems is hindered by the growing quantity and complexity of SoCs (System-on-a-Chip) and software stacks required for autonomous operation. Our study tackles two challenges: (1) fault handling in an autonomous driving system distributed across multiple processing cores and SoCs, and (2) isolation of multiple software modules consolidated in one SoC. To address the first challenge, we extend the state-of-the-art E-Gas layered monitoring concept. Similar to E-Gas, our safety mechanism has function, controller and vehicle layers. We propose to distribute these safety layers on processors with different ASILs (Automotive Safety Integrity Level). Besides, we implement self-test, fault injection and challenge-response protocols to detect faults at runtime in the safety mechanism itself. To facilitate distributed operation, our mechanism is built on top of the DDS (Data Distribution Service) software middleware for safety-critical embedded applications, as well as DDS-XRCE (eXtremely Resource Constrained Environment) for resource-constrained processor cores of the highest ASIL. To address the second challenge, our safety mechanism employs hardware-assisted hypervisors to isolate software modules and implement fail-silent behavior of faulty software stacks. We validate our safety mechanism on the NXP BlueBox hardware platform using the LG SVL simulator, Baidu Apollo software framework for autonomous driving, and Xen hypervisor. Our fault injection experiments demonstrate that the distributed safety mechanism successfully detects faults in an autonomous system and safely stops the vehicle when necessary.**

Keywords—autonomous vehicle, automated driving, safety, middleware software, hypervisor, fault injection, E-Gas, DDS, DDS-XRCE, Xen

I. Introduction

The automotive industry focuses on improving safety and comfort of the passengers while reducing energy consumption of the vehicles and air pollution. According to [1] most car accidents are attributed to human errors. Therefore, Autonomous Vehicles (AVs) are being designed to automate driving functionality and replace the human driver with reliable electronics and software. The degree of driving automation is categorized in five levels by the Society of Automotive Engineers [2]. Our study focuses on higher automation levels, where the vehicle is responsible for both lateral and longitudinal control as well as environment monitoring.

We distinguish two major trends in autonomous systems: computation distribution and consolidation. The *distribution* of computations is driven by the increasing algorithmic complexity of autonomous functionality supporting diverse driving scenarios and environments. No single SoC can execute all functionality at required performance, power and cost levels. Hence, we witness a growing quantity and complexity of the Electronic Control Units (ECUs), SoCs and software architectures for autonomous systems [3]. Indeed, [5] reports over a hundred ECUs in a modern vehicle and according to [4] cars run millions of lines of code, even without achieving fully autonomous operation. Each automotive SoC integrates an increasing number of general-purpose processing cores and application-specific compute engines, such as DSPs, GPUs and neural network accelerators. To distribute processing AV software, frameworks such as Baidu Apollo [6] and Autoware [10] rely on middleware. Middleware software stacks, such as Data Distribution Service (DDS) [11] and Robot Operating System (ROS) [12], provide dynamic service discovery and

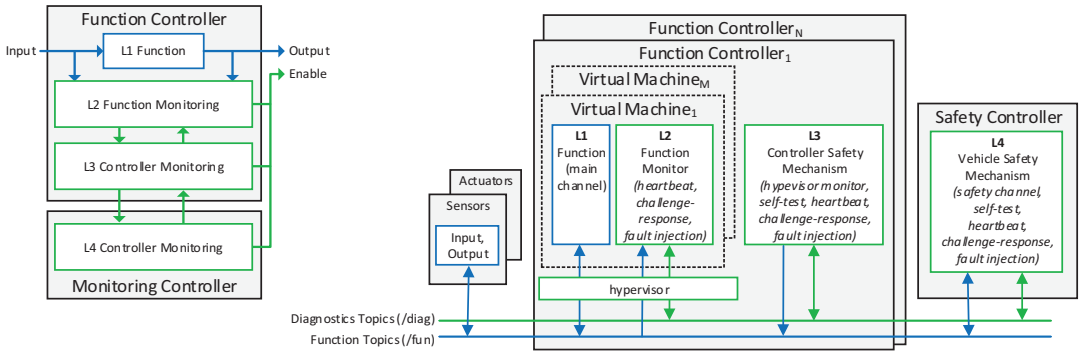

Fig. 1. Simplified E-Gas monitoring concept (left) and our Distributed Safety Mechanism concept (right).

seamless communication among distributed cores and SoCs. The computation distribution trend poses the challenge of handling faults in distributed processing.

Automotive system cost restrictions dictate *consolidation* (integration) of multiple functions on a single chip. Therefore, SoC and software developers are challenged with safe co-existence of several functions often with different safety requirements. Next to traditional process-level separation with Memory Management Units and OS-level separation on multicore processors, modern processor architectures, such as ARMv8, support running multiple OS stacks on software hypervisors using hardware virtualization support [9]. Hypervisors, also known as Virtual Machine Monitors, are capable of dynamically isolating and scheduling processing resources, memory and I/O devices [8]. They have been successfully used to safely consolidate, for example, performance-hungry infotainment applications with safety-critical instrument clusters software in automotive systems [7].

Safety-critical system developers adhere to *safety* standards, such as ISO 26262 [14], ISO/PAS 21448 [15], and ARINC 653 [31], describing safe development procedures, design patterns, fault models, safety mechanisms, etc. Furthermore, there exist state-of-the-art industry-accepted safety concepts, such as majority voting, built-in self-test and health monitoring [16]. Besides, fail-silent and fail-operational design patterns describe how a subsystem can fail without erroneous output or even continue its operation after a fault occurs [17].

Our study tackles two challenges in autonomous system design: (1) handling faults in distributed processing and (2) safe co-existence of consolidated software modules. This paper makes the following *contributions*:

1. a layered safety mechanism concept leveraging the middleware software to handle faults in an autonomous system with distributed and redundant processing;
2. a safety mechanism using hypervisors to enforce fail-silent behavior of complex software stacks;
3. a proof-of-concept evaluation of the first two contributions on automotive-grade hardware running a full AV software stack.

This paper continues with an overview of related work on safety concepts, middleware and hypervisors. In Section III we describe our distributed safety mechanism. Our experimental setup based on simulation and fault injection is covered in Section IV. Based on the conducted experiments we formulate conclusions in Sections V.

II. Related Work

The ISO 26262 safety standard introduces the concepts of freedom from interference and co-existence relevant to our study. Both concepts suggest isolation between consolidated functions, which is addressed in our study using hypervisors. Furthermore, the Automotive Safety Integrity Level (ASIL) decomposition in the standard requires independence among redundant elements contributing to the same safety goal. The distributed nature of our safety mechanism running on separate processors with different ASILs enables ASIL decompositions.

Our work originates from *E-Gas* [21] – a safety concept widely adopted in the automotive industry for engine control ECUs with the safety goal of avoiding unintended acceleration. The left diagram in Fig. 1 illustrates the E-Gas layered safety concept. The level 1 (L1) Function is monitored by the level 2 (L2) Function Monitor (FM), which can disable the Function output upon detecting a fault in L1 Function. Level 3 (L3) checks the Function Controller SoC and L2 FM, while level 4 (L4) monitors the Function Controller from outside. The original E-Gas concept labeled the external controller monitor L3, for simplicity of reference we call it L4.

Although E-Gas was originally devised for a single controller, [22] applied the layered concept to multicore processors and extended it to fail-operational redundant architectures. Our work analyzes applicability of E-Gas to a distributed processing architecture with multiple function controllers, jointly cooperating on the execution of autonomous driving. Furthermore, E-Gas was applied to processors with *hardware virtualization* support for hypervisors by [23], which benchmarked the virtualization technology against traditional microcontrollers with an external monitoring ASIC and multicore architectures. In contrast to [23] we use hypervisors not only to isolate the functional and safety components, but also to enforce fail-silent behavior of faulty software stacks.

There are also other safety mechanisms that have been applied to *distributed processing*. For example, the EcoTwin

truck platooning project [18] deployed several distributed health monitors reporting to a central arbiter that could switch vehicle control from the nominal to the safety channel when a malfunction is detected. Our study follows up on the EcoTwin research by evaluating hypervisors and mapping the application across automotive SoCs with different ASILs. Furthermore, [19] describes a distributed health monitoring system with virtualization using Adaptive AUTOSAR. However, they focus primarily on health monitoring on a multicore processor, omitting safety mechanism reactions and multi-SoC processing using software middleware, which we evaluated in our work.

Due to high complexity of the autonomous driving functionality, AV software frameworks split code in multiple software modules adhering to the Service-Oriented Architecture [20]. Inter-module communication and synchronization is carried out by a *software middleware*, providing service discovery, seamless data exchange among different platforms, Quality of Service (QoS), security, real-time guarantees, etc. Examples of middleware protocols include Cyber RT [24], DDS [11], ROS [12], and SOME/IP [25]. Noteworthy, DDS features real-time support, safety, security and good interoperability with other middlewares, such as ROS. In general, many of the middleware protocols adhere to the popular publish-subscribe pattern [13], where a sender does not send data to a concrete receiver, but publishes information on a common data bus. Other modules subscribe to the updates of the information on the bus. The data bus can be realized with an Ethernet network, on-chip shared memory, serial interface, etc. Our Distributed Safety Mechanism (DSM) reuses the middleware services to coordinate operation of safety components. Compared to [26], which also described a safety mechanism leveraging middleware and hypervisors, we added handling of internal faults in the DSM and studied different safety use-cases.

III. Distributed Safety Mechanism Concept

A centralized health monitor is simple from the architecture perspective. However, in distributed (autonomous) systems it suffers from high reaction time and low observability. By observability we mean the monitor's ability to detect faults and to read hardware and software state of the distributed components in sufficient detail. The DSM concept depicted on the right in Fig. 1 inherits the E-Gas layers for high observability and quick reactions. However, the E-Gas concept is applied to a single controller, whereas an AV has multiple function controllers labeled Function Controller$_1$ to Function Controller$_N$ in Fig. 1. Every Function Controller consolidates several Functions, which we isolated in virtual domains, labeled as Virtual Machine$_1$ to Virtual Machine$_M$. The blue components in Fig. 1, such as L1 Function, refer to the inherent autonomous functionality, while the elements of our DSM, such as L2 Function Monitor, are colored green. Note that in contrast to E-Gas, the DSM layers monitor distributed components in multiple Function Controllers and VMs. For distributed operation the overall system employs a *publish-subscribe middleware* software, communicating through Diagnostic Topics (/diag) and Function Topics (/fun). The input and output data of the function and safety control messages get published on the /fun topics, whereas diagnostics data are published on the /diag topics. Besides the Enable output of the E-Gas concept for the fail-silent behavior, our DSM includes a safety channel that can take over the vehicle control with situational awareness using sensor and function data from the /fun middleware topics. To ensure that only one publisher controls the vehicle, the middleware can use QoS features, such as strength (priority) in DDS [11] or the DSM can shut down all but one publisher. Note that the distribution of the /fun and /diag topics can be optimally realized using inter-chip or on-chip hardware. For example, communication between L2 and L3 can be kept on-chip, using shared memory, while L2 and L4 can communicate via Ethernet. The same safety mechanism concept applies to an integrated on-chip system, where the Function Controllers are integrated with a Safety Controller in a single package or die using different on-chip subsystems and cores.

The solid lines in Fig. 1 on the right denote physical inter-chip or on-chip boundaries, while the dotted lines refer to a virtually isolated domain, for example, by means of a hypervisor. The *hypervisor* traditionally isolates different function's software stacks including OSes and provides processor scheduling and memory allocation guarantees to safety-critical functions [4]. On top of that we propose to use hypervisors to implement a fail-silent behavior. If a faulty software stack yields wrong output, such as erroneous vehicle control commands, the DSM can detect it in one of its monitors and instruct the hypervisor using the /diag topics to pause the corresponding Virtual Machine (VM) domain. The silenced faulty function is then replaced by a degraded mode or a full-fledged redundant function in the DSM itself, which publishes on /fun topics. The latency of the domain pause must be low to respect the fault tolerance interval defined by the high-level safety goal. Note that the middleware traffic to and from the VMs in Fig. 1 goes through the hypervisor and hardware virtualization support for memory and resource management. Furthermore, the L3 Controller Safety Mechanism (CSM) monitors the hypervisor and runs in a SoC subsystem outside of hypervisor's control on an ARM Cortex-R or -M safety core.

Thanks to the distributed architecture, the DSM can handle hardware and software faults in various SoCs and ECUs. Various prior-art monitoring techniques can be integrated in DSM layers to support different *fault models*, such as a software program deadlock, a memory cell bit flip, and a short-circuit.

Furthermore, the DSM concept supports *redundancy* for fail-operational processing in autonomous driving. For example, L4 Vehicle Safety Mechanism (VSM) in Fig. 1 can integrate a safety channel, which takes over vehicle control when a fault is detected in the main channel of a Function Controller. Since L4 VSM has access to the function data on the /fun topics, it can even identify hazardous situations in the absence of faults discussed in ISO/PAS 21448 [15] and overrule the main channel actuations.

Using Fig. 2 in the following subsection A we discuss three scenarios, in which the DSM handles faults in the main autonomous Function denoted as blue in Fig. 1. Then we cover two scenarios with faults in the DSM itself in subsection B. Fig. 2 depicts simplified Finite State Machines (FSMs) of the DSM L2 FM, L3 CSM, and L4 VSM. The initial states are marked (INITIAL). The transitions labels have a prefix of the triggering element. For example, CSM: challenge in L2 FM indicates the challenge that L3 CSM sends to L2 FM. Multi-line transition labels refer to multiple events, which can independently trigger the state change. Furthermore, state actions under the state title refer to

middleware operations, such as publish or subscribe, and middleware topics, such as /diag for diagnostics and /fun for autonomous function data and control.

A. *Safety scenarios with a faulty function*

Scenario 1: AV function fault. L2 FM, monitoring L1 Function output on /fun, detects a problem in L1 Function and goes to the FUNCTION_FAULT state. In this state L2 FM publishes FAULT messages on /diag topics as long as L1 Function is faulty. Otherwise, it goes back to the FUNCTION_OK state. L3 CSM notices the FAULT on /diag topics and goes to the PAUSE_VM state along the FM: FAULT on /diag transition. Then it quickly pauses the faulty VM and publishes PAUSED to /diag. Triggered by the CSM: PAUSED on /diag transition, L4 VSM goes to SAFE_MANEUVER and publishes commands on /fun topics to safely maneuver the vehicle using sensor data on /fun topics to avoid collision.

This sequence of events exemplifies the high observability of L2 FM, a quick response by L3 CSM in the function controller and a safety maneuver carried out by an independent vehicle-level safety controller L4 VSM, which has access to the environment perception data on the /fun topics for vehicle control in diverse situations. The *safety maneuvers* that L4 VSM can command the vehicle to perform include a safe stop, an evasive maneuver or other vehicle-level safety actions. In the described scenario above, the software middleware enables seamless communication and coordinates safety-critical activities. Note that the hypervisor in this scenario does not only statically isolate the fault, but also actively silences the faulty software stack to limit its interference with the rest of the system.

Scenario 2: implausible /fun data. Thanks to the data distribution by the middleware, L4 VSM can periodically and independently check /fun data streams from L1 Function for plausibility. When such a check fails, L4 VSM follows the L1: function fault* transition to PAUSE_VM, instructing L3 CSM to pause the faulty VM by publishing the PAUSE request on the /diag topic. After L3 CSM's confirmation that the VM was paused, L4 VSM maneuvers the vehicle in SAFE_MANEUVER.

Scenario 3: VM failure. If a software module leaks memory or blocks a hardware resource, the whole VM can fail. The L3 CSM monitors VM using hypervisor diagnostics. For example, as soon as high CPU utilization is detected in a VM, the CSM: VM fault transition moves L3 CSM to PAUSE_VM to prevent fault propagation. Then L4 VSM takes over control of the vehicle.

B. *Safety scenarios with faults in the DSM*

The safety mechanism should also cope with its internal faults, which are called latent multi-point faults in the ISO 26262 [14]. There exist well-known techniques [16] to detect latent faults, such as built-in memory and logic self-tests, challenge-response protocols, runtime fault injections, and periodic heartbeats. Below we give two examples of how challenge-response protocols apply to our middleware-based DSM.

Scenario 4: DSM internal health status check. L4 VSM from Fig. 2 checks the health status of L3 CSM in one of the Function Controllers by publishing a challenge message on a /diag topic and starting a timer. The challenge can be, for example, computational to test the underlying processor core or storage-focused to test memory operation. L4 VSM measures the time that L3 CSM took to respond in order to detect missed real-time deadlines. If L3 CSM is healthy, it receives the challenge on the /diag topic, generates the response, and publishes it on the /diag topic on time. L4 VSM then receives the timely response and moves back to the VEHICLE_OK state. Otherwise it assumes that L3 CSM is faulty and moves to the SAFE_MANEUVER state. As shown in Fig. 2 L4 VSM can also check L3 CSM by instructing L2 FM to inject platform faults (e.g. high CPU usage) at runtime by publishing the INJECT_FAULT request on /diag.

Scenario 5: hypervisor monitor. Another noteworthy use case is L3 CSM's monitoring of the hypervisor, on which the Function Controllers run. According to the ISO 26262

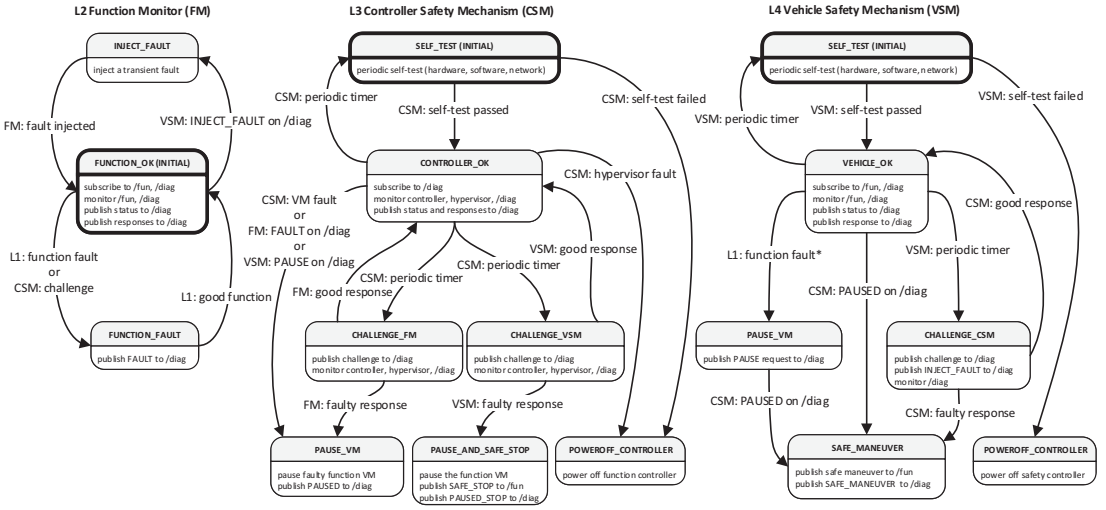

Fig. 2. Simplified DSM's Finite State Machines, where each transition is annotated with the origin level and related event.

terminology [14], a failing hypervisor is a Common Cause Fault, which can compromise the overall vehicle safety. To detect hypervisor faults, L3 CSM runs in a neighboring yet isolated safety subsystem outside of the hypervisor and continuously monitors hypervisor heartbeats using on-chip communication means such as shared memory or interrupts. When the missing hypervisor heartbeat is detected, L3 CSM powers down the Function Controllers by going to the POWEROFF_CONTROLLER state. Meanwhile the silence from the function domain(s) moves L4 VSM into the SAFE_MANEUVER state.

IV. Experimental Evaluation

To check feasibility of our DSM concept, we prototyped the DSM and conducted fault injection experiments.

A. Experimental setup with the DSM prototype

We prototyped the DSM concept in a *Hardware-In-the-Loop* (HIL) simulation setup. The setup integrates the LG SVL simulator [27] on a simulation PC and the Baidu Apollo 3.0 AV software stack [6] on the NXP BlueBox hardware prototyping platform [28] for autonomous driving, see Fig. 3. The LG SVL simulator models the environment and vehicles to provide sensor data to the Apollo software. Apollo processes sensor data as if they were obtained from real-life sensors and feeds actuation signals back to the simulator to drive the vehicle.

For compatibility with middleware-based AV software, we built the DSM on top of the *DDS middleware*. Due to resource constraints safety cores of the S32R274 SoC can run only a DDS subset, called DDS-XRCE. We built the L4 VSM on S32R274, using the open-source eProsima Micro XRCE-DDS [29], which connects a DDS-XRCE Client to DDS through a DDS-XRCE Agent, running on full DDS. In order to interlink different middlewares in our system, we built software bridges and integrated the DDS-XRCE Agent on LS2084. As to *virtualization* we used the Docker container engine and Xen hypervisor [30]. The Xen hypervisor supports spatial and temporal isolation as well as VM management operations, such as domain pause, which we used to realize the fail-silent behavior.

By mapping the DSM and AV software modules on SoCs according to their ASILs and resources in Fig. 3, our prototype implements *a subset of the DSM concept* from Fig. 1:

1. L1 Function is implemented by Apollo software modules. Apollo software modules run on quality managed SoCs, except for the vehicle control module allocated to a Xen hypervisor Domain U on the ASIL C processor S32V234.
2. L2 FM in Domain U reads L1 Function output from /fun and publishes diagnostics to /diag. To evaluate our DSM we disabled Apollo's safety mechanism.
3. L3 CSM in Domain 0 monitors and pauses Domain U if needed. It subscribes to /diag for requests or challenges from L4 VSM and publishes the status of hypervisor domains to /diag. Due to engineering complexity we did not run L3 CSM outside of Xen as defined in Section III.
4. L4 VSM is mapped on the ASIL D SoC S32R274. It can trigger L3 CSM to perform domain pause using /diag. The failover operation is carried out by L4 VSM publishing hard stop setpoints to /fun. The hard stop brakes the vehicle disregarding sensor data.

B. Latency measurements of pausing a virtual machine

The DSM must quickly silence a faulty virtual machine to prevent fault propagation. In our experiments the domain shutdown of the virtual machine took much more time compared to the domain pause operation. Hence, we use the Xen pause operation on the ASIL C S32V234 processor running the Null scheduler of the Xen hypervisor. The Null scheduler minimizes the scheduling overhead thanks to the static one-to-one mapping of virtual to physical cores. The pause latency includes the runtime of the libxl_domain_pause() function from Xen's libxl library and the overhead of the Yocto Linux in Domain 0 to start the pause operation triggered by an external middleware message. In 2000 experiments the arithmetic mean of the pause latencies was 55us, while the maximum outlier reached 1.5ms. Although this latency was low enough to silence a faulty VM within the fault tolerance interval of the safety goal to avoid collisions, the high outliers suggest the need for more deterministic scheduling in the hypervisor and VM kernels to carry out the VM pause.

C. Fault injection experiments to validate the DSM concept

To validate scenarios from Section III, we simulated road situations and injected faults into the system. A *driving scenario* is created in the LG SVL simulator, where the ego vehicle controlled by Apollo drives on a two-lane city road. In the

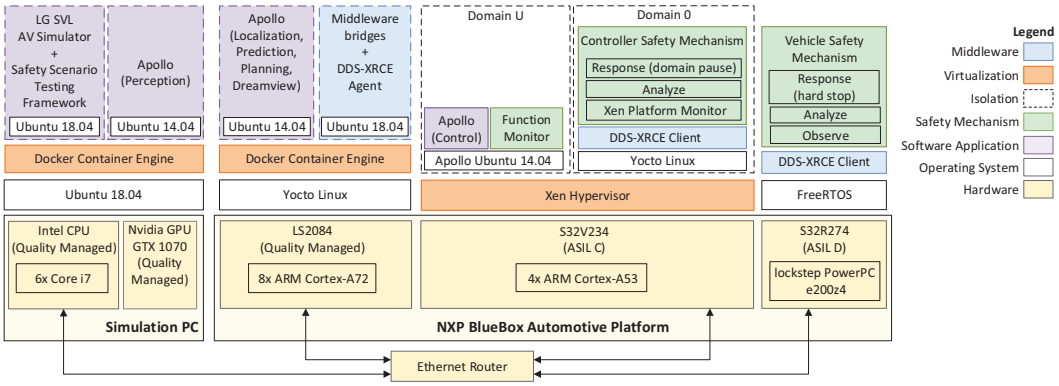

Fig. 3. Architecture of the Hardware-In-the-Loop experimental setup composed of a simulation PC and an automotive-grade platform for autonomous vehicles.

distance there is a stationary obstacle. To reach its destination ahead of the obstacle, the ego vehicle must change lanes.

First, we validated the most typical *Scenario 1* from Section III by injecting an invalid vehicle control message with acceleration actuation setpoint above the allowed 100%. The extended Function Monitor in Domain U from Fig. 3 reported this abnormal command to Control Safety Mechanism, which paused the Domain U VM, and Vehicle Safety Mechanism subsequently stopped the vehicle. Furthermore, we evaluated *Scenario 2* by injecting an invalid vehicle control message with brake and acceleration actuation setpoints simultaneously equal to 100%. Thanks to a simple check in the Vehicle Safety Mechanism, the vehicle was stopped right upon the detection of the fault. For *Scenario 3* we used the stress Linux tool to overload the CPU in Domain U, while the ego vehicle is changing lane. Consequently, the Apollo control module output was published at a significantly lower rate. Without the DSM, the ego vehicle started to wobble and eventually crashed into an obstacle on the road. With our DSM enabled, however, the Controller Safety Mechanism promptly paused the faulty VM and the Vehicle Safety Mechanism successfully performed a hard stop. Finally, we validated *Scenario 4* by killing the Controller Safety Mechanism in Domain 0. This was quickly noticed by the challenge-response protocol logic in the Vehicle Safety Mechanism, which then brought the ego car to a full stop without accidents.

V. Conclusion

To handle faults in distributed processing we designed a distributed safety mechanism based on the E-Gas layered safety concept and DDS publish-subscribe middleware. The DDS middleware protocol together with its subset, DDS-XRCE, enabled us to map the safety mechanism to resource-constrained processor safety cores with appropriate ASIL features. Furthermore, our DSM leveraged hardware-supported hypervisors to isolate faults and block propagation of failures. On top of traditional isolation, we used the hypervisor to pause the faulty software stack, implementing the fail-silent behavior useful for constructing large safety-critical systems. Besides handling faults in the autonomous functions, our mechanism deployed the following techniques to handle internal malfunctioning: challenge-response, self-test, and fault injection. The key aspects of the DSM concept were successfully evaluated on automotive SoCs in the NXP BlueBox HIL setup with the LG SVL autonomous driving simulator. Upon injection of an artificial fault into the system, the DSM safely stopped the simulated vehicle driven by the Baidu Apollo autonomous driving software stack.

References

[1] S. Singh, "Critical reasons for crashes investigated in the national motor vehicle crash causation survey," NHTSA, Washington, DC, USA, Rep. DOT HS 812 506, 2018.

[2] *Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems*, *SAE* Standard J3016_201401.

[3] K. Jo, J. Kim, D. Kim, C. Jang and M. Sunwoo, "Development of autonomous car—Part II: A case study on the implementation of an autonomous driving system based on distributed architecture," *IEEE Trans. Ind. Electron.*, vol. 62, no. 8, 2015, pp. 5119-5132.

[4] O. Burkacky, J. Deichmann, G. Doll, and C. Knochenhauer, "Rethinking car software and electronics architecture," McKinsey&Company, 2018.

[5] S. Tuohy, M. Glavin, C. Hughes, E. Jones, M. Trivedi and L. Kilmartin, "Intra-Vehicle Networks: A review," *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 2, 2015, pp. 534-545.

[6] Baidu Apollo, "Apollo open platform." Accessed September 2019. [Online]. Available: http://apollo.auto

[7] S. Karthik et al., "Hypervisor based approach for integrated cockpit solutions," in *Proc. IEEE 8th ICCE-Berlin*, 2018, pp. 1-6.

[8] A. Burns and R. I. Davis, "Mixed criticality systems – a review," 12th ed, Dept. Comput. Sci., University of York, 2019.

[9] R. Mijat and A. Nightingale, "Virtualization is coming to a platform near you," ARM, Version 8.0, 2011.

[10] S. Kato et al., "Autoware on board: enabling autonomous vehicles with embedded systems," in *Proc. 9th ACM/IEEE International Conference on Cyber-Physical Systems*, 2018, pp. 287-296.

[11] OMG, "DDS Foundation." Accessed September 2019. [Online]. Available: https://www.dds-foundation.org

[12] M. Quigley et al., "ROS: an open-source Robot Operating System," in *Proc. ICRA Workshop on Open Source Software*, 2009.

[13] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *Proc. 11th ACM Symposium on operating systems principles*, 1987, pp. 123-138.

[14] *Road Vehicles: Functional Safety*, *ISO* Standard 26262-1, 2018.

[15] *Road vehicles: Safety Of The Intended Functionality*, *ISO/PAS* Standard 21448, 2019.

[16] B. W. Johnson, "Fault-tolerant microprocessor-based systems," *IEEE Micro*, vol. 4, no. 6, 1984, pp. 6-21.

[17] A. Avizienis, J. Laprie, B. Randell and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable and Secure Computing*, vol. 1, no. 1, 2004, pp. 11-33.

[18] T. Bijlsma, M. Kwakkernaat and M. Mnatsakanyan. "A real-time multi-sensor fusion platform for automated driving application development," in *Proc. IEEE 13th International Conference on Industrial Informatics*, 2015, pp.1372-1377.

[19] M. Neukirchner, "Building performance ECUs with Adaptive AUTOSAR," 10th AUTOSAR open conference, USA, 2017.

[20] M. P. Papazoglou, P. Traverso, S. Dustdar and F. Leymann, "Service-Oriented Computing: state of the art and research challenges," Computer, vol. 40, no. 11, 2007, pp. 38-45.

[21] "Standardized E-Gas monitoring concept for gasoline and diesel engine control units," EGAS Workgroup, Version 6:57, viii, 2015.

[22] M. Grosmann, M. Hirz, and J. Fabian, "Efficient application of multi-core processors as substitute of the E-Gas (Etc) monitoring concept," in *Proc. SAI Computing Conference*, 2016, pp. 913-918.

[23] Y. Nakagawa, S. Arai, and R. Mariani, "Virtualization technology, and applying to E-Gas monitoring concept," FISITA 2014 World Automotive Congress, vol. 20, the Netherlands, 2015, pp. 86-92.

[24] Baidu Apollo, "Cyber RT Introduction." Accessed July 2019. [Online]. Available: https://github.com/ApolloAuto/apollo/tree/master/cyber

[25] SOME/IP, "Scalable service-Oriented MiddlewarE over IP (SOME/IP)." Accessed September 2019. [Online]. Available: http://some-ip.com

[26] P. J. van der Perk, "A distributed safety mechanism for autonomous vehicle software using hypervisors," M.S. thesis, Dept. Elect. Eng., Eindhoven University of Technology, the Netherlands, June 2019.

[27] LGSVL Simulator, "LGSVL simulator." Accessed September 2019. [Online]. Available: https://www.lgsvlsimulator.com

[28] NXP Semiconductors, "NXP BlueBox: Autonomous Driving Development Platform." Accessed September 2019. [Online]. Available: https://www.nxp.com/bluebox

[29] eProsima, "eProsima Micro XRCE-DDS." Accessed September 2019. [Online]. Available: https://micro-xrce-dds.readthedocs.io/en/latest/

[30] M. Raho, A. Spyridakis, M. Paolino and D. Raho, "KVM, Xen and Docker: a performance analysis for ARM based NFV and cloud computing," in *Proc. IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering*, 2015, pp. 1-8.

[31] *Avionics Application Standard Software Interface*, *ARINC 653* Standard, 1996.