Structure-Aware, Diagnosis-Guided ECU Firmware Fuzzing

QICAI CHEN, Fudan University, China
KUN HU, Fudan University, China
SICHEN GONG, Fudan University, China
BIHUAN CHEN*, Fudan University, China
ZIKUI KONG, ANHUI GuarDrive Safety Technology, China
HAOWEN JIANG, Fudan University, China
BINGKUN SUN, Fudan University, China
YOU LU, Fudan University, China
XIN PENG, Fudan University, China

Electronic Control Units (ECUs), providing a wide range of functions from basic control functions to safety-critical functions, play a critical role in modern vehicles. Fuzzing has emerged as an effective approach to ensure the functional safety and automotive security of ECU firmware. However, existing fuzzing approaches focus on the inputs from other ECUs through external buses (e.g., CAN), but neglect the inputs from internal peripherals through on-board buses (e.g., SPI). Due to the restricted input space exploration, they fail to comprehensively fuzz ECU firmware. Moreover, existing fuzzing approaches often lack visibility into ECU firmware' internal states but rely on limited feedback (e.g., message timeouts or hardware indicators), hindering their effectiveness.

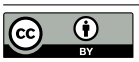
To address these limitations, we propose a structure-aware, diagnosis-guided framework, EcuFuzz, to comprehensively and effectively fuzz ECU firmware. Specifically, EcuFuzz simultaneously considers external buses (i.e., CAN) and on-board buses (i.e., SPI). It leverages the structure of CAN and SPI to effectively mutate CAN messages and SPI sequences, and incorporates a dual-core microcontroller-based peripheral emulator to handle real-time SPI communication. In addition, EcuFuzz implements a new feedback mechanism to guide the fuzzing process. It leverages automotive diagnostic protocols to collect ECUs' internal states, i.e., error-related variables, trouble codes, and exception contexts. Our compatibility evaluation on ten ECUs from three major Tier 1 automotive suppliers has indicated that our framework is compatible with nine ECUs. Our effectiveness evaluation on three representative ECUs has demonstrated that our framework detects nine previously unknown safety-critical faults, which have been patched by technicians from the suppliers.

CCS Concepts: • **Computer systems organization** → **Firmware**; • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: ECU Firmware Fuzzing, Serial Peripheral Interface, Diagnostic Feedback

*Bihuan Chen is the corresponding author.

Authors' Contact Information: Qicai Chen, School of Computer Science, Fudan University, Shanghai, China, qcchen23@m.fudan.edu.cn; Kun Hu, School of Computer Science, Fudan University, Shanghai, China, huk23@m.fudan.edu.cn; Sichen Gong, School of Computer Science, Fudan University, Shanghai, China, scgong24@m.fudan.edu.cn; Bihuan Chen, School of Computer Science, Fudan University, Shanghai, China, bhchen@fudan.edu.cn; Zikui Kong, ANHUI GuarDrive Safety Technology, Shanghai, China, kongzikui@guardrive.tech; Haowen Jiang, School of Computer Science, Fudan University, Shanghai, China, hwjiang23@m.fudan.edu.cn; Bingkun Sun, School of Computer Science, Fudan University, Shanghai, China, bksun21@m.fudan.edu.cn; You Lu, School of Computer Science, Fudan University, Shanghai, China, ylu24@m.fudan.edu.cn; Xin Peng, School of Computer Science, Fudan University, Shanghai, China, pengxin@fudan.edu.cn.



ACM 2994-970X/2025/7-ARTISSTA039
https://doi.org/10.1145/3728914

ACM Reference Format:
Qicai Chen, Kun Hu, Sichen Gong, Bihuan Chen, Zikui Kong, Haowen Jiang, Bingkun Sun, You Lu, and Xin Peng. 2025. Structure-Aware, Diagnosis-Guided ECU Firmware Fuzzing. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA039 (July 2025), 23 pages. https://doi.org/10.1145/3728914

1 Introduction

Modern vehicles contain many interconnected Electronic Control Units (ECUs), with the number varying from around thirty for low/mid-end cars to around hundred for high-end vehicles [59]. These ECUs manage a wide range of functions, from basic control functions like window control to safety-critical functions such as stability control [30], collision avoidance [29, 54, 56, 57] and airbag deployment [27]. While these ECUs enable sophisticated vehicle control and automation, their extensive deployment also raises concerns about functional safety and automotive security.

Functional safety, as defined in ISO 26262 [31], is the absence of unreasonable risk due to hazards caused by malfunctioning behavior of electrical/electronic systems. Such malfunctions can arise from systematic failures (due to specification errors, implementation flaws, or integration issues) or random failures (that occur stochastically in hardware). The automotive industry has witnessed several critical incidents resulting from violations of functional safety. For example, a software defect in Toyota's Electronic Throttle Control System (ETCS) caused unintended acceleration incidents between 2002 and 2010 due to an inadequate watchdog timer implementation [36]. A calibration error in Ford's Body Control Module and Powertrain Control Module caused unexpected vehicle stalls in 2021-2024 models [22]. A software defect in the transmission control unit (CVT-ECU) of 2019-2022 Mitsubishi Outlander Sport models triggered transmission failures and led to engine damage [42]. Therefore, it is important to comprehensively test ECUs to ensure their functional safety.

Automotive security, governed by standards like ISO/SAE 21434 [21], addresses the protection of vehicle systems against malicious attacks that could compromise safety, functionality, or data privacy. ECUs are vulnerable to cyber attacks due to their increasing connectivity and complexity. Several documented attacks have demonstrated their vulnerabilities. For example, researchers have demonstrated remote exploitation of ECUs through wireless interfaces [41], unauthorized manipulation of vehicle behavior through compromised ECUs [64], and extraction of sensitive information from ECU firmware [44, 45]. As vehicles become increasingly connected and autonomous, the potential impact of such security breaches grows more severe [13, 66]. Vulnerabilities in ECU firmware must be detected and patched during development, as post-deployment patches can be costly or even impractical [1, 19]. These security considerations, alongside functional safety requirements, necessitate testing techniques that can effectively evaluate both aspects of ECU firmware.

Fuzzing has become an effective approach to ensure the functional safety and automotive security of ECU firmware. Early effort primarily employs black-box fuzzing techniques [23, 35, 46, 48, 61, 63, 65, 68]. These techniques often rely on random mutations [23, 46, 63, 65] or structure-aware mutations [35, 48, 61, 68] of CAN messages, guided by observable ECU behaviors such as message timeouts and hardware indicators. To further improve effectiveness, grey-box fuzzing techniques [53] have been proposed to utilize control flow information to guide the mutation. However, existing techniques have two limitations. First, they primarily consider the inputs from external buses, particularly CAN, but neglect the critical inputs from on-board buses such as SPI. These on-board buses handle real-time interactions between the microcontroller unit (MCU) in ECUs and on-board peripherals [7, 37]. Due to the restricted input space exploration, existing techniques fail to comprehensively fuzz ECU firmware. Second, their feedback mechanism usually relies on external observable behaviors such as CAN messages timeouts or hardware indicators, providing limited visibility into the ECU's internal states. As a result, their feedback mechanism fails to effectively guide the mutation and thus hinders the effectiveness of fuzzing.

In the broader context of MCU firmware fuzzing, several approaches have been developed. SHIFT [40] runs instrumented firmware on MCUs to collect coverage information. GDBFuzz [17] utilizes hardware breakpoints to capture execution states without the need of code instrumentation. μAFL [38] leverages ARM's Embedded Trace Macrocell for detailed instruction tracing. However, these approaches are not readily applicable to ECU firmware fuzzing. The instrumentation required by SHIFT often exceeds the limited storage resource of ECUs. GDBFuzz's continuous breakpoint operations interfere with ECUs' strict timing requirements. μAFL's tracing demands specialized debugging hardware that is costly and difficult to integrate into the closed automotive ECU ecosystem.

To address the above limitations, we propose a structure-aware, diagnosis-guided framework, named EcuFuzz, to comprehensively and effectively fuzz ECU firmware. On the one hand, EcuFuzz simultaneously considers external buses (i.e., CAN) and on-board buses (i.e., SPI). This expanded input space enables the fuzzing of the ECU firmware's interaction with both vehicle networks and on-board ASICs, addressing a critical gap in existing approaches. To generate inputs, EcuFuzz leverages the structure of CAN and SPI to effectively mutate CAN messages and SPI sequences. To deliver inputs while meeting ECUs' real-time constraints, EcuFuzz implements a dual-core microcontroller-based peripheral emulator. This emulator enables precise timing control for SPI communication, allowing EcuFuzz to maintain the strict temporal requirements of ECUs during fuzzing.

On the other hand, EcuFuzz implements a new feedback mechanism to effectively guide the fuzzing process. EcuFuzz leverages ECUs' built-in unified diagnostic services (UDS) to access the ECUs' internal states, including error-related variables, trouble codes, and exception contexts. This internal visibility through ECUs' built-in UDS overcomes the limited visibility of existing approaches that only monitor ECUs' external behaviors, helps guide the fuzzing process toward potentially problematic states, and also facilitates technicians to detect and localize the potential faults.

To evaluate EcuFuzz, we first quantitatively analyze ten ECUs from three major Tier 1 automotive suppliers. EcuFuzz is compatible with nine ECUs that support both CAN and SPI interfaces, which demonstrates the practical usability for real-world ECUs. Then, we run EcuFuzz against each of the three representative ECUs, i.e., an Airbag Control Unit (ACU), a Front-Looking Camera (FLC), and a Front-Looking Radar (FLR) for 24 hours. Our fuzzing campaign reveals nine previously unknown safety-critical faults across these ECUs, including faults in airbag deployment control, radar obstruction detection, and sensor data processing. All the identified faults have been reported to the suppliers, and subsequently confirmed and patched by the technicians, which demonstrates the practical effectiveness of EcuFuzz in fault detection. We also run three state-of-the-arts against ACU where EcuFuzz detects seven faults. However, they fail to detect any fault.

In summary, this work makes the following contributions.

- We proposed a comprehensive and effective fuzzing framework, EcuFuzz, for ECU firmware. It expands input space by simultaneously targeting external buses and on-board buses, and leverages a new feedback mechanism through UDS to enable diagnosis-guided fuzzing.
- We implemented a dual-core microcontroller-based peripheral emulator. Its modular architecture enables adaptation to different on-board bus protocols through a simple interface, making EcuFuzz both cost-effective and extensible for automotive ECU firmware fuzzing.
- We evaluated EcuFuzz through experiments on real-world automotive ECUs, demonstrating its compatibility across different ECU hardware architectures and its effectiveness in discovering previously unknown safety-critical faults. Nine faults have been detected in three ECUs.

2 Background

ECUs Based on Classic AUTOSAR Architecture. Among various ECU architectures deployed in modern vehicles, this work focuses on ECUs based on Classic AUTOSAR architecture, which

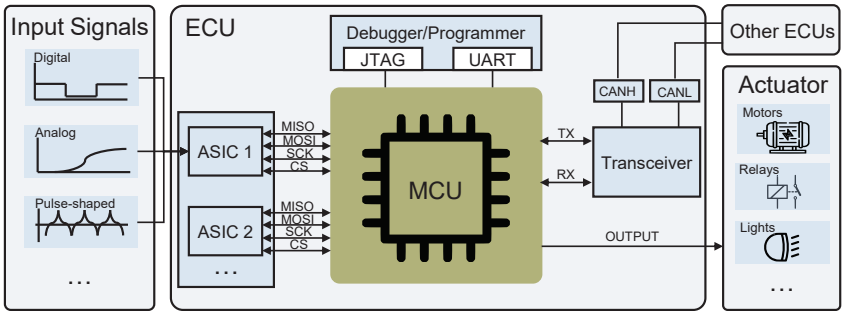


Fig. 1. ECU Hardware Architecture

are widely deployed in safety-critical control functions requiring high functional safety levels. AUTOSAR (AUTomotive Open System ARchitecture) provides a standardized software framework for automotive ECUs [5], helping manufacturers reduce hardware dependencies and simplify the integration of software components. The classic AUTOSAR architecture is divided into three layers, i.e., the application layer, the runtime environment (RTE) layer, and the basic software (BSW) layer. The application layer contains software components (SWCs) that implement an ECU's core functionalities. The RTE layer abstracts communication between SWCs and BSW, facilitating both intra- and inter-ECU data exchange. The BSW layer provides essential services such as memory management, diagnostics, and communication interfaces, ensuring smooth interaction between the application and external systems. It is worth mentioning that some high-end ECUs adopt System on Chip (SoC) architecture and run general-purpose operating systems like QNX. However, classic AUTOSAR ECUs are still widely deployed and particularly crucial as they directly handle safety-critical functions such as airbag deployment and brake control. Such ECUs operate under strict real-time and resource constraints, making their safety and security testing more challenging, compared to their high-end counterparts that already have a wide range of testing tools available.

ECU Hardware Architecture. Figure 1 illustrates a typical ECU hardware architecture, showing its main components and their interconnections. The core of an ECU is a microcontroller unit (MCU), which executes a firmware that contains AUTOSAR software components and basic software, etc. This firmware is typically developed using AUTOSAR tools, and flashed into the MCU's non-volatile memory by debugging interfaces. The MCU connects to external vehicle networks via transceivers that convert differential signals into digital signals for MCU processing, enabling communication with other ECUs. Moreover, the MCU connects to application-specific integrated circuits (ASICs) through on-board buses. ASICs, sometimes referred to as System Basis Chips (SBCs) [11], provide support functions for ECU operation. Specifically, ASICs process and condition various input signals, including digital (discrete on/off signals), analog (continuous voltage levels), and pulse-shaped signals (complex waveforms), before forwarding them to the MCU. ASICs also integrate other support functions such as power management, system supervision, and wake-up logic. This design of ASICs reduces overall system complexity and allows the MCU to focus on core computational tasks.

From the perspective of an ECU firmware, its inputs primarily originate from two sources, i.e., external buses (e.g., CAN) carrying network traffic from other ECUs, and on-board buses (e.g., SPI) transmitting pre-processed sensor data from the ASICs. This architecture enables ECUs to efficiently handle both system-wide coordination and local sensor processing tasks, making them capable of performing complex control functions while maintaining real-time performance requirements.

Serial Peripheral Interface (SPI). SPI is a widely used synchronous serial communication protocol for short-distance communication between MCUs and peripherals [24]. It operates in a master-slave configuration, typically with the MCU as the master. The SPI protocol uses four signal

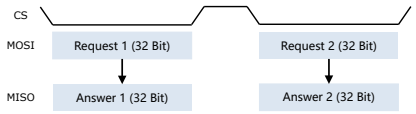


Fig. 2. In-Frame SPI Communication

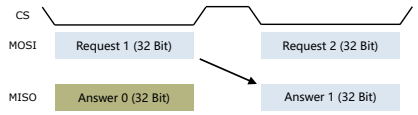


Fig. 3. Out-Frame SPI Communication

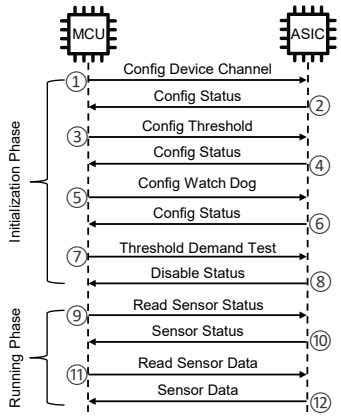


Fig. 4. Communication Process Between the MCU and the On-Board ASIC in the ECU via SPI

	Time [s]	Chip Select	MOSI	MISO
①	0.029374	ASIC	0x6838013C	0x20200007
②	0.037554	ASIC	0x6818013C	0x202C0091
③	0.030374	ASIC	0x68780128	0x20200007
④	0.030874	ASIC	0x68580128	0x202C0091
⑤	0.062026	ASIC	0x6160C80C	0x20600034
⑥	0.070205	ASIC	0x6140C80C	0x20606402
⑦	0.078386	ASIC	0x04400218	0x20600005
⑧	0.291006	ASIC	0x0D40000C	0x60600700
⑨	0.299186	ASIC	0x3200001C	0x2800C00E
⑩	0.305706	ASIC	0x33000014	0x2800001B
⑪	0.312246	ASIC	0xC0000008	0x2A1FFF0A
⑫	0.318786	ASIC	0xC0000008	0x2A1FFE89

Fig. 5. An Examaple of a SPI Record between the MCU and the On-Borad ASIC in the ECU

lines for synchronous communication, Master Output Slave Input (MOSI) for data transmission from master to slave, Master Input Slave Output (MISO) for data transmission from slave to master, Chip Select (CS) for selecting and enabling the target slave device, and Serial Clock (SCK) for synchronizing data transfer between devices. This configuration enables full-duplex communication, with the master device (MCU) controlling the clock signal and slave selection through the SCK and CS lines while exchanging data with the slave device (ASIC) through the MOSI and MISO lines. In automotive ECUs, SPI communication often involves 32-bit data words, with the Most Significant Bit (MSB) transmitted first. Two common SPI communication modes used in ECUs are as follows.

- **In-Frame Mode.** As illustrated in Figure 2, the request and answer are interleaved within a single communication cycle. The master sends a 32-bit request over the MOSI line, and the slave immediately responds with a 32-bit answer over the MISO line within the same communication cycle. This mode requires the slave to process the request and generate a answer in real-time.
- **Out-Frame Mode.** As illustrated in Figure 3, the request and answer occur in two successive communication cycles. The master sends a 32-bit request in one communication cycle, and the slave responds with a 32-bit answer in the subsequent communication cycle. This mode allows a delay between the request and answer, providing more processing time for the slave.

Communication Between MCU and ASIC. Figure 4 provides the communication process between the MCU and the ASIC via SPI, while Figure 5 illustrates an example of a SPI record captured during the communication process. The SPI record contains a sequence of entries, and each entry contains 32-bit digital signals transmitted on MOSI and MISO lines, along with the corresponding chip select signals that identify the target ASIC. Based on the ASIC technical manual, these signals can be decoded into specific instructions during both initialization phase and running phase.

During the initialization phase, the MCU configures the ASIC's device channels, thresholds, and watchdog settings, with each configuration followed by a status check. A threshold demand test is then performed, where the MCU verifies the ASIC's ability by properly comparing sensor values

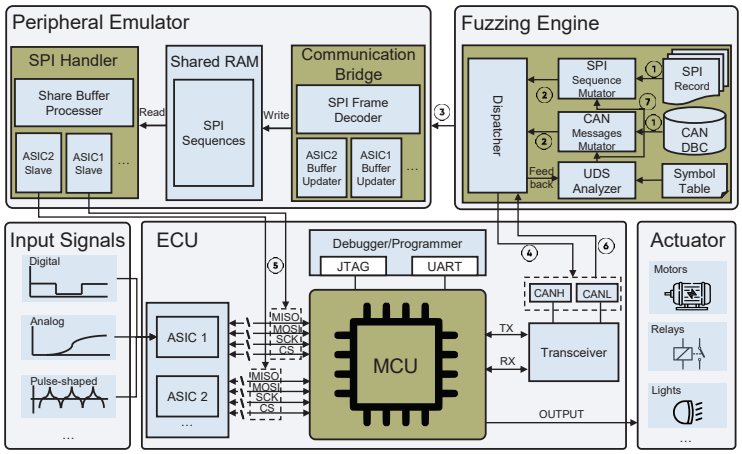


Fig. 6. Framework Overview of EcuFuzz

against configured thresholds. This test must be passed for safety-critical functions (e.g., airbag deployment) to properly operate. Once initialized, the MCU enters its running phase, where the MCU cyclically reads sensor status and data from the ASIC. For example, in an airbag control unit, the ASIC continuously processes acceleration signals from crash sensors before sending the processed data to the MCU via SPI for airbag deployment decisions.

Controller Area Network (CAN). CAN is the predominant protocol for inter-ECU communication in vehicles [12]. It employs a message-based structure with an arbitration ID and data payloads up to 8 bytes. CAN's key features include fixed data rates, prioritized message transmission, and robust error detection via a 16-bit Cyclic Redundancy Check (CRC). These characteristics make CAN ideal for real-time automotive applications, enabling efficient exchange of sensor data, control commands, and diagnostic information across vehicle systems.

Unified Diagnostic Services (UDS). UDS is a standardized automotive diagnostic protocol, defined in ISO-14229 [32], to facilitate communication between diagnostic tools and ECUs in vehicles. It provides a unified framework for various diagnostic services, enhancing vehicle serviceability and enabling sophisticated diagnostic capabilities [34]. A key feature of UDS is the ability to retrieve Diagnostic Trouble Codes (DTCs), which are standardized codes that indicate specific faults or malfunctions in the vehicle. For example, DTC B0001 represents an airbag deployment circuit failure. Through UDS, technicians can access these DTCs, read real-time sensor data, reprogram ECU modules, and execute specific diagnostic routines such as sensor calibration.

3 Approach

We first introduce the framework overview of EcuFuzz, and then elaborate on each key module.

3.1 Overview

We design EcuFuzz as a structure-aware, diagnosis-guided fuzzing framework to test the firmware of automotive ECUs. ECUs are input-triggered, and they undergo state transitions and computations based on inputs from external and on-board buses. At its core, EcuFuzz uses this characteristic by adopting an input-centric testing paradigm, i.e., systematically crafted inputs are injected to elicit a wide range of ECU behaviors, allowing for the identification of potential defects within the firmware. The framework overview of EcuFuzz is presented in Figure 6, and its workflow algorithm is shown in Algorithm 1. Overall, EcuFuzz is composed of two components, i.e., a fuzzing engine hosted on a PC workstation, and a peripheral emulator implemented on a dual-core microcontroller. To enable

Algorithm 1 The Workflow of EcuFuzz

Require: *ECU*: the target ECU under fuzz; *ELF*: the binary file of the ECU firmware; *SPIRecord*: the SPI communication record file; *DBC*: the CAN database file; *N*: the maximum number of iterations
Ensure: FaultReports: a set of detected faults with triggering inputs

```
1:  SPISequences ← ExtractSPISequences(SPIRecord)                    ▷ See Section 3.2
2:  CANMessages ← ExtractCANMessages(DBC)                            ▷ See Section 3.3
3:  SPISeeds ← SPISequences, CANSeeds ← CANMessages
4:  PeripheralEmulator ← InitializePeripheralEmulator()              ▷ See Section 3.4
5:  SymbolTable ← ExtractSymbolTable(ELF)          ▷ Extract variable names and addresses
6:  UDSAnalyzer ← InitializeUDSAnalyzer(SymbolTable)                 ▷ See Section 3.5
7:  for i ← 1 to N do
8:      ECUState ← UDSAnalyzer.GetECUState()
9:      MSPI ← SPISequenceMutator(SPISeeds, ECUState)                ▷ See Section 3.2
10:     MCAN ← CANMessageMutator(CANSeeds)                           ▷ See Section 3.3
11:     Dispatcher.SendSPISequences(MSPI)
12:     Dispatcher.SendCANMessages(MCAN)
13:     PeripheralEmulator.SendSPISequences()                        ▷ See Section 3.4
14:     errorVars ← UDSAnalyzer.MonitorErrorVariables()              ▷ See Section 3.5
15:     dtcs ← UDSAnalyzer.MonitorDTCs()                             ▷ See Section 3.5
16:     exceptions ← UDSAnalyzer.MonitorExceptions()                 ▷ See Section 3.5
17:     if errorVars ≠ ∅ and (dtcs ≠ ∅ or exceptions ≠ ∅) then
18:         FaultReport ← GenerateFaultReport(errorVars, dtcs, exceptions, MSPI, MCAN)
19:         FaultReports ← FaultReports ∪ {FaultReport}
20:         if dtcs ≠ ∅ then
21:             UDSAnalyzer.ClearDTCs()                              ▷ See Section 3.5
22:         end if
23:     end if
24:     if new error-related variables are covered then
25:         SPISeeds ← SPISeeds ∪ {MSPI}, CANSeeds ← CANSeeds ∪ {MCAN}
26:     end if
27: end for
28: return FaultReports
```

the fuzzing of the ECU firmware, we disconnect the SPI connections between the MCU and on-board ASICs in the ECU under fuzz. This disconnection allows our peripheral emulator to take the place of the original on-board ASICs and simulate their behaviors.

The workflow of EcuFuzz begins with extracting seed inputs from the *SPI record* and *CAN DBC* files ① (Line 1–2). The SPI record contains pre-recorded SPI communication sequences between the MCU and on-board ASICs, while the CAN DBC defines the type and format of CAN messages. We respectively parse SPI record and CAN DBC to extract SPI sequences and CAN messages. The extracted SPI sequences and CAN messages serve as the seed corpus for our mutation-based fuzzing (Line 3). Then, the *SPI Sequence Mutator* and *CAN Message Mutator* generate mutated inputs based on seed inputs ② (Line 9–10). Through the *Dispatcher*, the mutated SPI sequences are first encoded into CAN messages and then transmitted to the *Peripheral Emulator* ③ (Line 11), whereas the mutated CAN messages are directly transmitted to the ECU ④ (Line 12). The *Peripheral Emulator*, implemented on a dual-core microcontroller, plays a critical role in simulating ASIC behaviors. Specifically, the *Communication Bridge* receives the encoded CAN messages, and decodes them back into SPI sequences. The decoded SPI sequences are written to the *Shared RAM*. The *SPI Handler* manages real-time SPI communication through slave modules by reading SPI sequences from the *Shared RAM* and transmitting them to the MCU ⑤ (Line 13). Then, the *UDS Analyzer* collects the ECU's internal states as the

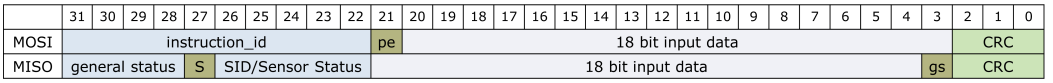


Fig. 7. Structured 32-Bit SPI Frame Format for MOSI and MISO in MCU-ASIC Communication

feedback ⑥ (Line 14–16). It inspects error-related variables based on the *Symbol Table* extracted from the binary file of the ECU firmware (Line 5), monitors active DTCs, and analyzes exception contexts. If a potential fault is detected based on the feedback, a fault report is generated for technician to analyze, and resume the ECU to normal running phase (Line 17–23). If new error-related variables are covered (i.e., the input triggers new error states), the input is preserved for subsequent mutation ⑦ (Line 25). Finally, when the campaign is finished, a set of fault reports is returned.

3.2 SPI Sequence Mutator

We first introduce the generation of the seed corpus of SPI sequences, and then present our structure-aware mutation strategy and context-aware mutation scheduler.

3.2.1 SPI Sequence Seed Generation. Since ECUs follow a fixed power-up sequence, we use a logic analyzer [52] to capture the complete SPI communication between the MCU and on-board ASICs from ECU power-up through initialization phase until reaching running phase. The logic analyzer records the raw 32-bit digital signals (i.e., SPI frames) transmitted on MOSI and MISO lines, along with the corresponding chip select signals that identify the target ASIC for each transmission. These recorded communication traces constitute a complete SPI Record, as illustrated in Figure 5. The initialization phase contains a deterministic sequence of SPI frames for ECU configuration, while the running phase consists of cyclically repeated SPI frames for sensor data processing. By capturing this entire operational cycle, we obtain both the initialization SPI sequence and running SPI sequence with their complete set of SPI frames in the correct order, which are sufficient to serve as the seed corpus for mutation-based fuzzing as only the field values within these SPI frames need to be mutated.

As shown in Figure 7, the on-board ASIC defines a specific SPI frame format for SPI communication, where each SPI frame on the MOSI and MISO lines follows a fixed 32-bit structure. The request frame on the MOSI line consists of fields including instruction identifier, status flags, data, and CRC. Similarly, the response frame on the MISO line has its own 32-bit structure with corresponding fields. Each entry in the SPI record is defined as a tuple $L = \langle t, cs, m_{out}, m_{in} \rangle$, where t is the timestamp, cs is the chip select signal, m_{out} and m_{in} are the 32-bit SPI frame on MOSI and MISO respectively. To parse each entry in the SPI record, we analyze the on-board ASIC technical manual to understand the SPI communication protocol, and create a JSON-formatted SPI instruction definition file. Each instruction in this definition file defines a pair of the MOSI and MISO frames as a tuple $I = \langle inst, \{f^1_{mosi}, ..., f^n_{mosi}\}, \{p^1_{mosi}, ..., p^n_{mosi}\}, \{f^1_{miso}, ..., f^k_{miso}\}, \{p^1_{miso}, ..., p^k_{miso}\}, \{r^1_{miso}, ..., r^k_{miso}\} \rangle$, where $inst$ is the instruction identifier in the MOSI frame (i.e., bits 22-31 as shown in Figure 7), f^i_{mosi} and f^j_{miso} are the name of the i-th and j-th field in the MOSI and MISO frames respectively, p^i_{mosi} and p^j_{miso} specify the bit positions of each field, and r^j_{miso} is the valid value range for the j-th MISO field. Notice that only MISO fields are subject to mutation since they represent ASIC responses that we aim to fuzz, and the MOSI fields are not mutated because they represent MCU requests.

Based on this definition file, we develop a parser to parse each entry in the SPI record. For each entry L, the parser first extracts the instruction identifier from bits 22-31 of the MOSI frame. Using this instruction identifier as the key, the parser looks up the corresponding instruction tuple I in the definition file, and parses MOSI and MISO frames according to I by extracting values from their specified bit positions (i.e., p^i_{mosi} and p^j_{miso}) and associating them with their corresponding field names (i.e., f^i_{mosi} and f^j_{miso}). Through this parsing process, we obtain a sequence of MISO frames for

two distinct phases. Notice that MOSI frames are not included because they are not mutated during fuzzing but are used for correctly locating the MISO frames during parsing. Specifically, in the running phase, specific SPI frames with fixed instruction patterns repeat cyclically, such as those for sensor data reading and diagnostic checking. By observing these repeated instruction patterns, we can identify a sequence of MISO frames belonging to the running phase. Consequently, the non-repeated sequence of MISO frames that appear from power-up until the first occurrence of these repeated patterns belongs to the initialization phase, as it occurs only once during power-up. The extracted SPI sequences from the initialization and running phases serve as the seed corpus, and will be mutated to test ECU startup behavior and cyclic operation behavior.

3.2.2 Structure-Aware Mutation Strategy. Building upon the instruction tuple I defined for SPI frame parsing, we propose a structure-aware mutation strategy that systematically applies different mutation operators to different MISO fields based on their functional roles and constraints. Formally, we define a mutation tuple M by extending I, i.e., $M = \langle I, \{O^1_{miso}, ..., O^k_{miso}\}\rangle$, where O^j_{miso} denotes the mutation operator pool for the j-th MISO field. Each mutation operator pool consists of the following three types of mutation operators.

- **AFL's Standard Mutation Operators.** These operators include basic bit-level mutations such as bit flips and arithmetic operations. They are primarily assigned to MISO fields that represent state information, where each bit corresponds to a specific status or flag (e.g., sensor status flags). By targeting individual bits using the bit positions p^j_{miso}, they help to effectively test the ECU firmware's capability to handle unexpected or corrupted state indicators. For example, flipping a bit that indicates a sensor's active status can test the error handling capability for various sensor states.
- **Edge-Case Mutation Operators.** These operators generate values that lie just outside the defined valid ranges of specific MISO fields. They are specifically targeted at MISO fields that represent data fields with clearly defined valid ranges (i.e., r^j_{miso} from I). For example, for a resistor value field in a seat occupancy sensor, edge-case operators might generate values slightly below the minimum or above the maximum thresholds. They help to test the ECU firmware's capability to correctly validate sensor data and handle unexpected or extreme input values. By referencing the bit positions p^j_{miso}, these mutations ensure that only the relevant bits corresponding to the data field are mutated, maintaining the integrity of other unrelated fields.
- **Range-Constrained Mutation Operators.** These operators produce values within the defined boundaries of each MISO field, as specified by r^j_{miso}. They can be assigned to any MISO fields, regardless of their functional role, since they ensure that the generated values remain within valid ranges. They help to test the ECU firmware's behavior under various valid but uncommon ASIC responses. For example, simulating sustained high acceleration values during prolonged bumpy road conditions can test the capability to handle extended periods of intense sensor data inputs. These operators utilize the valid ranges r^j_{miso} to generate appropriate values, and apply them to the correct bit positions p^j_{miso} to maintain data integrity.

During fuzzing, only one mutation operator is selected from the operator pool O^j_{miso} for each MISO field and applied per fuzzing iteration. This strategy enables comprehensive fuzzing of ASIC responses while preserving the SPI frame structure defined in the ASIC technical manual. For example, when fuzzing an acceleration sensor data field in a MISO frame, our strategy may employ range-constrained operators to evaluate the ECU firmware's behavior under different but valid acceleration values, or leverage edge-case operators to verify boundary checks. After mutation operations, a CRC recalculation mechanism ensures the mutated frames maintain valid CRC values based on algorithms from the ASIC technical manual. This prevents mutated frames from being discarded at the protocol level due to CRC validation failures, allowing them to reach ECU's processing logic.

3.2.3 Context-Aware Mutation Scheduler. To further enhance the fuzzing effectiveness while maintaining SPI communication integrity, we propose a context-aware mutation scheduler that applies different mutation operators during different ECU's operational phases. Specifically, the scheduler monitors the ECU's internal state through UDS (as will be described in Section 3.5), which enables it to determine whether the ECU is in initialization phase or running phase. Based on this phase information, appropriate mutation operators are dynamically selected as follows. During the initialization phase, the mutation process selectively applies operators from the pool O^{j}_{miso} to each MISO field. Specifically, it disables mutation operators that could affect protocol-critical MISO fields such as status flags so as to ensure the proper completion of the ECU startup process. In contrast, during the running phase, the full mutation operator pool is enabled for most MISO fields. This allows comprehensive testing of sensor data and status fields in MISO frames.

3.3 CAN Message Mutator

Similar to our SPI sequence mutator, our CAN message mutator utilizes a structure-aware mutation strategy based on CAN DBC file. While some existing approaches attempt structure-aware mutation through reverse engineering of CAN messages, they often fail to satisfy AUTOSAR end-to-end (E2E) protection checks [4]. Instead, we use CAN DBC file to obtain precise message definitions, enabling the generation of valid inputs that can pass E2E protection checks while exploring potential defects.

To generate a seed corpus for CAN messages, we use `cantools` [9] to parse DBC file and extract CAN message definitions. A CAN message includes message attributes (i.e., ID, name, and data length code (DLC)), signal characteristics (i.e., start bit, length, byte order, and scaling parameters), and timing parameters (i.e., transmission cycle time and timeout values). For each CAN message type in the DBC file, we generate multiple seed CAN messages by constructing a message frame with the specified ID and DLC, then populating each signal field with values from its valid physical range. These values are converted to raw binary format using signal-specific scaling parameters before being placed in their designated bit positions. Multiple seeds are generated with value combinations of different signal fields, including minimum, maximum, and typical operating values, to ensure a relatively diverse coverage of the CAN message exploration space.

Our mutation strategy preserves critical message attributes while mutating signal values. For each signal field within a CAN message, our strategy follows three steps. First, it extracts the raw binary value from the specified bit positions defined in the DBC file. Second, it converts this raw value to its physical representation using the DBC-specified scaling parameters (e.g., factor and offset). Third, it applies mutation operators that are specifically assigned based on the signal's characteristics as defined in the DBC file. Similar to our SPI sequence mutation, different types of operators are assigned based on the signal's functional role, i.e., bit-level operators for status flags, edge-case operators for threshold-sensitive values, and range-constrained operators for continuous sensor data. These mutated values are converted back to raw binary format and carefully inserted into their correct bit positions, maintaining the specified byte order to ensure message integrity. For example, for a vehicle speed message type (ID: 0x123, DLC: 8 bytes) containing a speed signal (bits: 8-20, scaling factor: 0.05625, valid range: 0-240 km/h), a seed message might represent a typical speed of 60 km/h. Our mutation strategy might apply range-constrained operators to test normal speed variations, or edge-case operators to verify boundary handling at the 240 km/h limit, while preserving the message's structural integrity as defined in the DBC file.

Our CAN message mutator operates in parallel with our SPI sequence mutator, enabling simultaneous fuzzing of both external and on-board buses. This integrated approach is crucial for detecting defects that only manifest under specific combinations of inputs. For example, during our ACU fuzzing, we identified airbag deployment threshold defects that emerged only when mutated CAN messages

carrying collision prediction data coincided with specific SPI-transmitted acceleration sensor data. These defects would remain undetected if fuzzing each bus in isolation, demonstrating the importance of concurrent mutation across different input buses.

3.4 Peripheral Emulator

While external bus simulation is well-supported by tools like CANoe [26] or PEAK-CAN [25], simulating on-board ASIC communication for fuzzing presents a timing challenge. Specifically, on-board ASIC communication, particularly via SPI, requires precise timing and real-time processing capabilities. Automotive SPI communication in In-Frame mode (see Section 2) operates at a rate up to 10 Mbaud, requiring request processing and response generation within the same communication cycle. For the fuzzing purpose, these responses must be dynamically generated by our mutation strategy while maintaining protocol timing. Any delay can corrupt the entire communication frame.

To address this timing challenge, we first considered FPGA-based solutions, which can support SPI communication timing requirements through parallel processing capabilities [51]. However, FPGA-based solutions present practical challenges for implementing peripheral emulation. While FPGAs excel at parallel processing, they require precise hardware-level implementation of communication protocols, making the design and verification process inherently complex. Debugging timing and protocol-related issues in hardware is more challenging than software-based approaches. Moreover, FPGA designs are relatively rigid once implemented, requiring hardware redesign to support different ASIC protocols or adapt to protocol variations. These limitations make FPGA-based solutions less practical for our peripheral emulation needs.

Instead, we adopt a protocol-centric approach that leverages a key characteristic of modern ECU architecture, i.e., all peripheral inputs are preprocessed by on-board ASICs before being transmitted to the MCU through standardized protocols like SPI. This architectural design enables our emulator to simulate diverse peripherals by capturing and replaying the standardized SPI communication sequences between ASICs and MCU, without directly interfacing with physical sensors. Such a protocol-centric approach significantly reduces testing costs, compared to traditional Hardware-in-the-Loop (HiL) testing that requires expensive peripheral-specific simulators, while requiring only minimal modifications to the SPI frame parsing logic to support new peripherals.

We implement this protocol-centric design using a dual-core microcontroller architecture that separates real-time SPI communication from mutated inputs management. This design provides the required timing precision while maintaining flexibility for fuzzing adaptation. The peripheral emulator is composed of the following three modules.

Communication Bridge. The communication bridge manages the interface between the fuzzing engine and SPI communication. It receives encoded SPI sequences via CAN messages from the fuzzing engine, decodes them through the SPI frame decoder, and writes them to the shared memory through the corresponding ASIC buffer updater. Due to the CAN frame size limitation, we implement message fragmentation and reassembly mechanisms for transmitting complete SPI sequences. This fragmentation mechanism employs sequence numbers and acknowledgments to ensure reliable transmission of mutated SPI sequences between the fuzzing engine and the communication bridge. Once received, these SPI sequences are reassembled before being written to the shared memory. In our implementation, this module runs on the Cortex-M4 core of the STM32H755 microcontroller.

Shared RAM. The shared memory interface between the communication bridge and the SPI handler utilizes a circular buffer structure, enabling continuous updates of SPI sequences without interrupting real-time SPI communication. To prevent race conditions in shared memory access, we implement a hardware-level synchronization mechanism through the STM32H755's AXI-SRAM. We prioritize SPI interrupts on the Cortex-M7 core to maintain deterministic communication timing.

SPI Handler. Dedicated to real-time SPI communication, the SPI handler manages communication with the MCU through ASIC slaves. Each ASIC slave simulates a specific on-board ASIC, implementing timing and protocol requirements for In-Frame and Out-Frame communication modes. We implement these slaves using the Cortex-M7 core of the STM32H755 microcontroller, leveraging its high-performance characteristics to meet strict timing requirements. To ensure uninterrupted data flow, we implement a double-buffering mechanism with interrupt-driven callbacks. When an SPI transfer completes, an interrupt triggers the buffer switch and prepares new data for the next transfer, maintaining continuous communication without timing violations.

3.5 UDS Analyzer

Our UDS analyzer monitors the ECU's internal state as the feedback for fuzzing by leveraging built-in UDS capabilities. Traditional instrumentation-based and coverage-based feedback approaches become impractical and ineffective for automotive ECUs for two reasons, i.e., the limited resources of MCUs (typically with flash memory and RAM in the range of a few megabytes and hundreds of kilobytes) and the cyclical nature of ECU operations where similar code paths are repeatedly executed for sensor data processing and control actions. Instead, our UDS analyzer aims to provide deep visibility into the ECU's internal state without requiring code instrumentation or specialized hardware.

To this end, we initially consider existing tools such as `python-uds` [10]. However, due to maintenance issues and unresolved bugs in these tools, we implement our own UDS client that constructs diagnostic request messages and processes diagnostic responses according to ISO 14229 standard, utilizing `python-can` [58] for CAN message transmission and reception. This implementation supports essential UDS services including memory read (`0x23`), DTC operations (`0x19`, `0x14`), and diagnostic session control (`0x10`), enabling systematic monitoring of ECU's internal state through standardized diagnostic interfaces [32]. Specifically, our feedback mechanism via UDS analyzer employs a three-level monitoring strategy, ranging from basic state tracking to critical fault detection. The first level (Section 3.5.1) is designed to guide the fuzzing process, while the second level (Section 3.5.2) and the third level (Section 3.5.3) are designed to detect potential faults.

3.5.1 Inspecting Error-Related Variables. Our UDS analyzer extracts error-related variable information from the ECU's ELF file by parsing its symbol table, identifying variables that indicate error conditions through naming patterns (e.g., variable names containing "error", "invalid" and "fault"). Using the UDS `0x23` service, our analyzer first reads these error-related variables when the ECU is in a DTC-free state to establish their baseline values. During fuzzing, it continuously monitors these variables through the UDS `0x23` service. When the values of these variables deviate from their baselines, potential error conditions are indicated. Moreover, our analyzer compares the set of deviated variables with those deviated in previous inputs. If new variables show deviations (i.e., variables that have not deviated in previous inputs), the corresponding input is added to the seed corpus for subsequent fuzzing iterations. This feedback serves to guide the mutation process.

3.5.2 Monitoring Diagnostic Trouble Codes. Our UDS analyzer leverages standardized DTCs to detect and track firmware anomalies. Using the UDS service `0x19`, it periodically polls the ECU for active DTCs, which provide standardized indicators of specific faults according to ISO 14229-1 standard [32]. After recording, our analyzer clears the detected DTC using the UDS service `0x14` to prevent it from interfering with future inputs, ensuring each input starts from a known clean state. This DTC-based monitoring together with the variable state analysis provides standardized fault indicators that are specifically designed for automotive diagnostics, easing manual fault analysis.

3.5.3 Analyzing Exception Contexts. Our UDS analyzer also captures severe system faults that may disrupt normal UDS communication. The AUTOSAR's `ErrorHook` and `ProtectionHook` functions

record critical errors (e.g., memory access violations, and privilege escalation attempts) in the Non-Volatile Memory (NVM) module [6]. When severe faults trigger ECU resets through the watchdog timer, our analyzer uses the UDS service `0x23` to retrieve these persistent error records from NVM after system recovery. This mechanism ensures that critical faults causing system-level failures are captured, even when they temporarily disable UDS communication, thereby completing the comprehensive monitoring strategy with severe fault detection capabilities.

3.5.4 False Positive Filtering. As required by the ASPICE standard [3], automotive suppliers conduct functional validation testing before ECU deployment, where one key aspect is to verify diagnostic capabilities by deliberately sending specific input patterns. These DTC-triggering input patterns are recorded in validation testing logs, which we can obtain from the suppliers. During fuzzing, when our mutated inputs accidentally match these known input patterns, the resulting DTCs represent the ECU's expected diagnostic behavior rather than potential firmware faults. Therefore, to reduce such false positives, we develop a filtering mechanism based on known DTC-triggering input patterns from functional validation testing. It identifies whether mutated inputs contain the same critical bits that are intentionally used in functional validation testing to verify specific diagnostic capabilities.

Specifically, let I_f represents the set of inputs that trigger a specific DTC during fuzzing, and I_t represents the set of inputs that trigger the same DTC during functional validation testing. For each input $i_f \in I_f$, if there exists an input $i_t \in I_t$ such that $i_f \,\&\, i_t = i_t$ (where & denotes the bitwise AND operation), meaning that i_f actually contains all the critical bits in i_t that are used to verify this diagnostic capability in functional validation testing, i_f is considered to trigger a false positive and is thus filtered; otherwise, i_f is considered to trigger a potential fault.

4 Evaluation

We have implemented EcuFuzz with 14.2K lines of C and Python code. To evaluate the compatibility and effectiveness of EcuFuzz, we conducted experiments to answer three research questions.

- **RQ1 Compatibility Evaluation**: How compatible is EcuFuzz with ECU hardware architectures?
- **RQ2 Effectiveness Evaluation**: How effective is EcuFuzz in detecting faults in ECU firmware?
- **RQ3 Ablation Study**: How do the two key components in EcuFuzz impact its effectiveness?

To answer **RQ1**, we used ten diverse ECUs from three major Tier 1 automotive suppliers who are our collaborators, and quantitatively assessed whether EcuFuzz is compatible with the ECU hardware architectures together with the technicians from the suppliers. To answer **RQ2**, we ran EcuFuzz against three representative ECUs, i.e., an Airbag Control Unit (ACU), a Front-Looking Camera (FLC), and a Front-Looking Radar (FLR), for 24 hours. We also compared EcuFuzz with three state-of-the-arts, i.e., SecFuzz [23], AutoFuzz [48] and EffCAN [53], using ACU. To answer **RQ3**, we compared EcuFuzz's effectiveness with/without UDS-based feedback and structure-aware mutation using ACU.

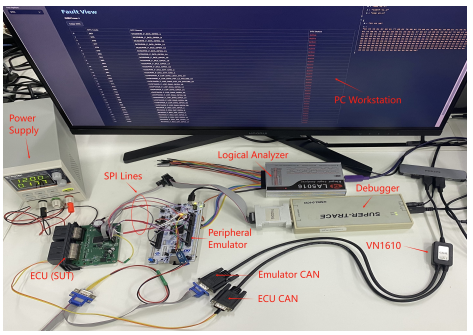


Fig. 8. Hardware Configuration in Our Evaluation

Our experiments were conducted on a Lenovo R9000K laptop equipped with an AMD Ryzen 9 5900HX CPU, 32 GB of RAM, running Windows 11. Figure 8 illustrates the hardware configuration in our evaluation. It included a Vector VN1610 interface for CAN communication, a Lauterbach debugger for firmware flashing, and a Zeroplus LAP-C logic analyzer (LA5016) for capturing SPI record. A NUCLEO-H755ZI-Q development board was used as the peripheral emulator.

Table 1. Statistics of the Hardware Architectures of Ten Representative ECUs

ECU	MCU Family	MCU Model	On-Board ASIC	SPI?	CAN?	Supplier
Airbag Control Unit (ACU)	Infineon AURIX	TC234LP	CCL1600B	Yes	Yes	Supplier A
Front-Looking Camera (FLC)	Infineon AURIX	TC397XP	XAZU3EG-1SFVC784Q	Yes	Yes	Supplier A
Front-Looking Radar (FLR)	NXP	S32R274	AWR2944	Yes	Yes	Supplier A
Engine Control Unit (ECU)	NXP	MPC5777C	UJA1169	Yes	Yes	Supplier B
Transmission Control Unit (TCU)	Infineon AURIX	TC277	TLE9278BQX	Yes	Yes	Supplier B
Body Control Module (BCM)	NXP	S12VR64	MC33972	Yes	Yes	Supplier B
Anti-lock Braking System (ABS)	STMicroelectronics SPC5	SPC560B54	L9374	Yes	Yes	Supplier B
Electric Power Steering (EPS)	Infineon AURIX	TC275	TLE9180D-31QK	Yes	Yes	Supplier B
Battery Management System (BMS)	TI C2000	TMS320F28027	MC33772B	Yes	Yes	Supplier C
Electronic Control Suspension (ECS)	Renesas RH850	R7F7015803AFP-C^KA3	None	None	Yes	Supplier C

4.1 Compatibility Evaluation (RQ1)

Table 1 reports the statistics of the hardware architectures of the ten representative ECUs from three suppliers (whose names are anonymized due to confidentiality agreements). Overall, these ECUs represent a diverse range of control systems found in modern vehicles, encompassing different functionalities and various MCU architectures [18, 47].

Specifically, these ECUs employ diverse MCU architectures, ranging from high-performance processors like Infineon AURIX TC397XP with multiple cores for complex ADAS functions, to cost-effective processors like TI C2000 TMS320F28027 for basic control tasks. Each MCU model is paired with specific on-board ASICs tailored to their functions; e.g., CCL1600B works with TC234LP in ACU for acceleration data processing, while AWR2944 works with S32R274 in FLR for radar signal processing. ECS stands out as an exception, using Renesas RH850 R7F7015803AFP without an on-board ASIC. This architectural difference stems from its specific functional requirements. ECS primarily controls pneumatic suspension systems, managing four electromagnetic valves for air spring adjustment by direct GPIO connections. This relatively simpler control topology allows the MCU to directly interface with actuators without an intermediary on-board ASIC for complex signal processing. This variety in MCU-ASIC combinations demonstrates the diversity of ECUs' hardware architectures.

The majority (i.e., nine out of ten) support both SPI and CAN interfaces. Therefore, EcuFuzz's design, targeting both external (CAN) and internal (SPI) communication buses, is compatible with these nine ECUs supporting both SPI and CAN. This compatibility indicates that EcuFuzz can provide testing capabilities for a wide range of automotive ECUs across different functionalities, hardware architectures, and suppliers. While the current implementation of EcuFuzz focuses on CAN and SPI interfaces, the modular architecture of peripheral emulator allows for adaptation to other protocols.

4.2 Effectiveness Evaluation (RQ2)

4.2.1 Evaluation on Three Real-World ECUs. While we analyzed ten ECUs from three suppliers in our compatibility evaluation, we conducted in-depth effectiveness evaluation on three representative ECUs from Supplier A, i.e., ACU, FLC and FLR. This focused evaluation was driven by two practical considerations. First, supplier A provided more comprehensive technical documentation and debugging support, enabling thorough analysis of the fuzzing results. Second, these three ECUs represent different safety integrity levels and functional domains, i.e., ACU for passive safety (ASIL D), and FLC and FLR for active safety (ASIL B), each implemented on different MCU architectures (TC234LP, TC397XP and S32R274 respectively). The MCU of ACU processes acceleration data from on-board ASICs to determine airbag deployment. The MCU of FLC receives information like detected objects and lane lines from on-board ASICs, and completes basic Level 2+ advanced driver assistance tasks. The MCU of FLR processes raw signals from the on-board ASICs, applying clustering and classification algorithms to identify object information from the original radar reflections.

Table 2. Effectiveness Results of EcuFuzz on Three Real-World ECUs

ECU	# Exec.	# Exec. Triggering DTCs	# Unique DTCs	# Filtered DTCs	# Exceptions	# Faults
ACU	43,120	2,637	287	63	1	7
FLC	42,830	1,340	104	17	8	1
FLR	38,131	497	59	21	5	1

Table 2 shows the results after running EcuFuzz for 24 hours against each ECU. Overall, approximately 40,000 input executions were generated per ECU. ACU, with the highest safety integrity level (ASIL D), exhibited the highest number of executions (i.e., 2,637) that triggered DTCs, resulting in 287 unique DTCs triggered. By applying the false positive filtering mechanism in Section 3.5.4, these were further filtered into 63 unique DTCs for manual investigation, effectively eliminating false positives from expected diagnostic behaviors. Despite having the highest number of DTCs, ACU only triggered one exception in one execution. On the contrary, FLC and FLR (ASIL B) triggered a much lower number (i.e., 17 and 21) of DTCs, but a higher number (i.e., 8 and 5) of exceptions. This inverse relation between ASIL levels and exception numbers aligns with the ISO 26262 requirement that higher ASIL levels require more rigorous fault detection mechanisms. Such mechanisms detect and handle potential faults earlier in the processing chain, generating DTCs rather than allowing issues to escalate into more severe system-level exceptions. Besides, ACU has more sensors and actuators than FLC and FLR for the diagnostic routines to monitor, resulting in more complex diagnostic routines and consequently a higher number of triggered DTCs.

We reported the filtered DTCs and exceptions, together with the inputs that triggered them, to the technicians of supplier A. They manually analyzed them and detected seven, one and one previously unknown faults in ACU, FLC and FLR, respectively, demonstrating the effectiveness of EcuFuzz.

4.2.2 Case Studies on the Detected Faults. We analyzed these nine safety-critical faults, which have been confirmed and patched by supplier A. Our analysis of these faults reveals that they are often triggered by edge cases involving complex interactions between multiple CAN and SPI inputs, i.e., scenarios that are challenging to identify through supplier's functional validation testing.

NVM Data Loss during Power-Down. In normal power-down scenarios, ACU relies on a capacitor (nominally charged to 12V) to supply power for critical operations, including writing to Non-Volatile Memory (NVM). This capacitor ensures data integrity even if the main power supply is compromised during a collision. However, we discovered a fault that, if the capacitor voltage approaches 9V before ACU initiates the power-down sequence, there may be insufficient time for all modules to complete their NVM write operations. This fault stems from ACU's threshold for NVM write operations (set to 9V) and time required for various modules to record their state.

Task Priority Inversion in Deployment Logic. We revealed a priority inversion fault in ACU firmware's deployment logic. It occurs when a medium-priority CAN message processing task holds a mutex that protects the deployment threshold parameters, while a high-priority task, processing acceleration data from the ASIC, needs to access these parameters. If a low-priority diagnostic task becomes active during this period, it can prevent the medium-priority task from releasing the mutex, blocking the high-priority task from updating critical deployment parameters. This scenario was triggered when EcuFuzz simultaneously sent specific CAN messages and SPI sequences that caused intensive parameter updates. The firmware failed to implement a proper mutex protection mechanism in this critical path, potentially affecting airbag deployment timing.

Self-Test State Lock during Initialization. During the initialization phase, we identified a fault related to the self-test process of the CCL1600B chip. ACU configures CCL1600B parameters via SPI communication, including sensor channel settings, thresholds, and filter coefficients. Subsequently, CCL1600B initiates a self-test to verify its sensor data reading and threshold comparison capabilities.

However, we observed when sensor data exceeds the set threshold during the self-test, ACU becomes locked in the self-test state, preventing ACU from entering its running phase.

Undefined Retry Timeout Mechanism. During initialization, ACU configures the CCL1600B chip via SPI communication. After configuring the registers, MCU reads back register values to determine if the chip has correctly written the configuration values. If the written values do not match read-back values, it performs a retry. However, if the number of retries is not properly set, the chip is faulty, causing ACU to be stuck in an infinite loop trying to configure the registers.

Unchecked Watchdog Timer Interval. During initialization, ACU configures watchdog timer's timeout period. If the MCU exceeds this period without properly resetting the watchdog timer, the watchdog will reset the MCU. If the watchdog timeout is set to a negative value, the watchdog cannot function correctly, causing the MCU to become stuck in an infinite loop and unable to reset.

Airbag Unintended Deployment Anomaly. In scenarios involving active and passive system fusion, the active safety system sends dynamic and static objects around the vehicle to ACU via CAN bus to adjust the deployment threshold, enabling ACU to deploy the airbag more accurately. In special scenarios (e.g., collisions with trucks), the acceleration threshold is set lower to achieve slightly earlier airbag deployment. However, lowering the threshold should not cause unintended airbag deployment when the vehicle is driving on bumpy roads. We found a fault where unintended airbag deployment occurred on bumpy roads while fuzzing both the CAN and SPI simultaneously.

Time-Critical Data Processing Race Condition. We revealed a timing fault in ACU's crash detection logic. When the mutated acceleration data from CCL1600B indicates potential crash events, it triggers complex crash analysis algorithms. Meanwhile, if the mutated CAN messages contain collision prediction data suggesting high crash probability, both data streams require immediate processing. However, due to improper task scheduling in the firmware, the collision prediction processing could delay the acceleration data analysis beyond the designated 1ms time window, causing ACU to miss the optimal airbag deployment timing. This fault is critical in complex crash scenarios where both pre-crash and crash data analysis are essential for optimal deployment.

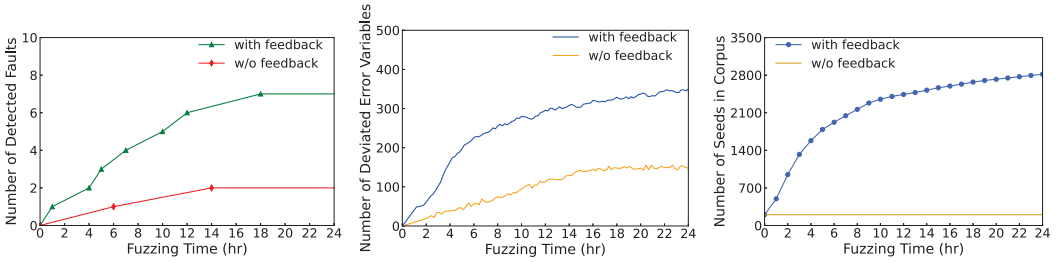
Unhandled Object Types in MCU. During the object detection process, FLC's on-board ASIC sends SPI frames containing object information to the MCU. We detected a fault where setting an unknown object type in these SPI frames caused the Autonomous Emergency Braking (AEB) function to fail. Specifically, when the MCU encountered an unfamiliar object type, it failed to trigger the AEB system properly. This fault could lead to potential collisions in real-world driving scenarios, as the vehicle might not respond appropriately to obstacles on the road.

Radar Obstruction False Positive. FLR erroneously reports radar obstruction when two conditions coincide, the on-board ASIC reports no reflection points via SPI, and the CAN bus indicates the vehicle is moving. This false positive occurs because the firmware fails to properly correlate the lack of radar data with vehicle speed. It could lead to unnecessary deactivation of ADAS features.

4.2.3 Comparison with State-of-the-Art. We conducted comparative experiments only on ACU for two reasons. First, ACU contains extensive diagnostic functions monitoring 32 pyrotechnic circuits, 16 RSUs and 32 switches, having rich error-related variables and DTCs suitable for evaluating the effectiveness of fuzzing. Second, ACU's Warning Lamp signals transmitted in CAN messages align with the hardware indicators required by the state-of-the-arts. However, FLC and FLR do not provide such signals, making the comparison infeasible. To the best of our knowledge, EcuFuzz is the first work that can efficiently fuzz both external buses and on-board buses for ECU firmware. We selected three state-of-the-arts for comparison, i.e., SecFuzz [23], AutoFuzz [48], and EffCAN [53], due to their diverse methodologies, which cover the spectrum from pure black-box to grey-box fuzzing strategies and collectively represent the current state-of-the-arts in ECU fuzzing research.

Table 3. Comparison Results of EcuFuzz with Three State-of-the-Art Approaches

Approach	# Exec.	# Exec. Triggering DTCs	# Unique DTCs	# Warning Lamp	# Exceptions	Error-Var Coverage	# Faults
ECUFuzz	43,120	2,637	287	32	1	86%	7
SecFuzz	43,050	42	8	9	0	8%	0
AutoFuzz	43,085	31	5	7	0	6%	0
EffCAN	43,100	56	12	15	0	12%	0



(a) Fault Discovery Analysis (b) Error-Related Variable Analysis (c) Corpus Growth Analysis

Fig. 9. Impact of Our UDS-Based Feedback Mechanism on EcuFuzz's Effectiveness

To ensure a fair comparison, we enhanced these approaches by equipping them with CAN message structure knowledge through DBC files, ensuring they had equal capabilities in understanding CAN message structures during mutation. Moreover, to enable comprehensive comparison metrics, we equipped these approaches with our UDS analyzer to collect various feedbacks for the purpose of comparison. As we lacked their specific hardware setups, we also modified them to monitor the Warning Lamp signals in CAN messages that controlled their hardware indicators. It preserved their original intention of monitoring ECU's external behaviors while adapting to our environment.

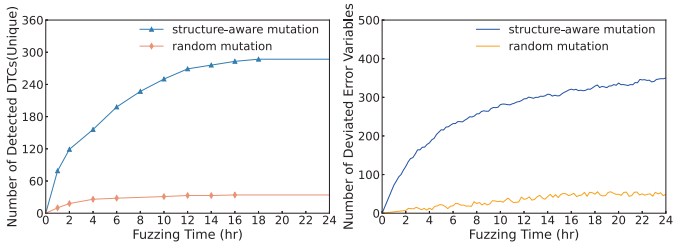
Table 3 reports the comparison results. While the three state-of-the-arts generated a similar number of input executions as EcuFuzz, they triggered a significantly fewer number of DTCs and Warning Lamp activations, while achieving a significantly lower coverage of error-related variables. This effectiveness gap can be attributed to two main factors. First, AUTOSAR-compliant ECUs implement multi-layered validation for CAN messages, including CAN identifier and DLC validation at the driver level, and E2E protection checks at the COM layer. These validations make it challenging for the three approaches which focus solely on fuzzing CAN messages to trigger ECU anomalies. Second, more importantly, these three approaches only focus on CAN inputs, while modern ECUs process inputs from both CAN and on-board buses. In ECU firmware, CAN-related code and its associated error-related variables only account for a small portion, as most functional modules are dedicated to processing sensor data and control logic via on-board buses. By simultaneously fuzzing both CAN and SPI inputs, EcuFuzz can access error-related variables across both communication and functional modules, leading to a significantly higher coverage of error-related variables.

4.3 Ablation Study (RQ3)

To evaluate the impact of our two key components, i.e., our UDS-based feedback mechanism and our structure-aware mutation strategy, we conducted two ablation studies on ACU.

4.3.1 Impact of UDS-Based Feedback. To evaluate the impact of our UDS-based feedback, we compared the effectiveness of EcuFuzz with and without our feedback for 24-hour fuzzing period.

Fault Discovery Analysis. Figure 9a illustrates the number of faults discovered over the fuzzing time. With our feedback mechanism enabled, EcuFuzz identified seven faults during the 24-hour



(a) Unique DTCs Analysis (b) Error-Related Variable Analysis

Fig. 10. Impact of Our Structure-Aware Mutation Strategy on EcuFuzz's Effectiveness

period. In contrast, with our feedback mechanism disabled, EcuFuzz detected only two of the same faults, demonstrating the effectiveness of UDS-based guidance in exploring diverse error states.

Error-Related Variable Deviation Analysis. Figure 9b shows the number of error-related variables deviated from their baseline values over the fuzzing time. With our feedback mechanism enabled, we observed a steady increase in the number of deviated variables, particularly rapid in the initial four hours. Our feedback mechanism maintained a consistently higher number of deviated variables by directing mutations towards inputs that were more likely to trigger new error states. The slowing growth rate after the initial four hours suggests that EcuFuzz has explored many of the readily accessible error states, with further exploration requiring more sophisticated input combinations. In contrast, with our feedback mechanism disabled, EcuFuzz shows a lower number of deviated variables. It reflects the random nature of mutations without the targeted guidance. As the ECU's error handling mechanism requires sustained error conditions to confirm faults, random mutations might intermittently trigger and then inadvertently resolve errors in subsequent cycles. This process leads to fluctuations in the number of deviated variables, as errors might be temporarily induced and then inadvertently reset due to subsequent random inputs. After approximately 16 hours, the growth in the number of deviated variables plateaus, although oscillations continue. This plateau is attributed to the latching mechanism employed for critical error-related variables. Once these variables change state, they are latched and preserved in non-volatile memory (NVM) for subsequent diagnostic purposes, even if the error-inducing conditions are no longer present. These latched errors persist unless cleared through specific UDS routine control commands, contributing to the sustained number of deviated variables despite ongoing random mutations.

Corpus Growth Analysis. Figure 9c shows that our feedback mechanism drove rapid corpus growth in the first 8 hours, increasing to approximately 2,200 seeds. The growth rate then gradually decreased, with the corpus size reaching around 2,800 seeds by the end of the 24-hour period. In contrast, without feedback, the corpus size remained constant at the initial size, as no new error-triggering inputs were preserved. This difference correlates with the error-variable deviation trend, reflecting the progressive discovery of error-triggering inputs through feedback guidance.

4.3.2 Impact of Structure-Aware Mutation. To evaluate the impact of our structure-aware mutation, we compared EcuFuzz with and without structure-aware mutation over a 24-hour fuzzing period. Without structure-aware mutation, EcuFuzz performs random mutations.

Unique DTCs Analysis. Figure 10a shows that with structure-aware mutation, EcuFuzz detected a higher number of unique DTCs (i.e., 287) and identified seven faults, while random mutation only detected 31 unique DTCs and identified zero fault. This difference is attributed to the mutation strategy's capability to generate structurally valid inputs that pass low-level driver validations, allowing the inputs to reach higher-level application logic where more complex DTCs are triggered. In contrast, random mutation typically generates inputs that fail basic validation checks (e.g., CRC validation, and valid bit checks), resulting in a limited set of approximately 30 driver-level DTCs.

Error-Related Variable Deviation Analysis. As shown in Figure 10b, structure-aware mutation achieved a higher number of deviated error-related variables. Random mutation primarily triggered changes in driver-level error-related variables, maintaining a relatively constant number of around 50 deviated variables throughout the fuzzing period.

In summary, these results demonstrate the significant contribution of our UDS-based feedback mechanism and structure-aware mutation strategy to the achieved effectiveness of EcuFuzz.

4.4 Threats to Validity

Internal Validity. Our work faces three main threats to internal validity. First, our false positive filtering mechanism relies heavily on functional validation testing logs. If these logs are incomplete or do not capture all intended diagnostic behaviors, legitimate diagnostic responses could be incorrectly flagged as faults. Second, our identification of error-related variables depends on naming conventions in symbol tables. It may miss relevant variables if they do not follow expected naming conventions, potentially affecting the comprehensiveness of our error state monitoring. Third, the modifications we made to baselines, including adding CAN message structure knowledge, equipping them with our UDS analyzer for feedback collection, and adapting their hardware indicator monitoring mechanism to use CAN messages, could affect their original behavior. However, these modifications should theoretically enhance rather than impair their performance by providing them with better input structure understanding and more comprehensive monitoring capabilities.

Construct Validity. Our evaluation relies on the number of triggered DTCs, exceptions, and error-related variable deviations as indicators of potential faults. While these indicators align with standardized automotive diagnostic practices, they may not capture every form of functional anomaly. For example, the firmware might exhibit performance or real-time scheduling issues without raising immediate diagnostics. Additionally, our framework assumes that legitimate firmware faults eventually map to detectable DTCs or manifest through specific error variables and exception hooks. If an ECU's diagnostic coverage is incomplete or does not robustly log intermittent faults, some issues might remain undetected. Extending our framework to correlate timing metrics and resource usage with abnormal conditions could enhance the construct validity of our evaluation.

External Validity. EcuFuzz relies on three key components, i.e., the STM32H755 development board, the binary ELF file of the ECU firmware, and the CAN DBC files. While the STM32H755 is widely available, access to ELF and DBC files is usually restricted to Tier 1 suppliers or trusted partners of original equipment manufacturers (OEMs). Importantly, EcuFuzz does not require access to source code, relying solely on ELF and DBC files, which are relatively accessible for academic institutions collaborating with automotive industry partners. However, our framework's applicability remains primarily within in-house testing or collaborative projects. The current implementation of EcuFuzz is compatible only with ECUs that support both CAN and SPI communication. ECUs using alternative protocols, such as LIN or FlexRay, or those without on-board ASICs, are not currently supported. However, the underlying methodology of EcuFuzz can be adapted to these other protocols with appropriate modifications due to our modular design. Additionally, while our evaluation covered ECUs from three major suppliers, there might be architectural variations in ECUs from other suppliers that could affect tool compatibility. Our future work is focused on extending our fuzzing framework by exploring more specific ECU firmware characteristics and potentially expanding the compatibility to include a wider range of communication protocols.

5 Related Work

ECU Security and Safety Testing. Early automotive fuzzing efforts primarily employed black-box techniques, focusing on CAN bus communication. In terms of random mutation-based approaches,

Fowler et al. [23] used random mutations of CAN messages, monitoring vehicle responses via hardware indicators. Werquin et al. [63] extended it by combining multiple fuzzing strategies with automated physical response detection. Nyamdelger et al. [46] performed fuzzing by feeding CAN messages generated by a real vehicle. Yeo et al. [65] combined CAN message fuzzing with sensor-based feedback monitoring in a simulated vehicle environment. Then, structure-aware mutation-based approaches emerged. Patki et al. [48] focused on fuzzing UDS requests by analyzing message structures. Kim et al. [35] analyzed CAN message logs to generate more effective inputs. Varghese et al. [61] refined it with automated reverse engineering-guided fuzzing, incorporating real-time ECU response analysis. Zhang et al. [68] further enhanced structure-aware fuzzing by combining bit flip rate analysis with generative adversarial networks to produce realistic CAN messages. To improve fuzzing effectiveness through better feedback mechanisms, grey-box approaches were developed. Radu et al. [53] designed EffCAN, utilizing control flow graph coverage of ECU firmware to guide payload mutations. More recently, Dunne et al. [14] employed reverse engineering to guide CAN bus fuzzing, but limited details are uncovered in their short paper. Varghese et al. [60] used powertrace monitoring to detect system responses in their black-box CAN bus fuzzer. In addition to fuzzing, hardware-in-the-loop (HiL) simulation has been widely used for ECU functional safety testing [28, 33, 43], and it has evolved to include fault injection techniques [8, 55] for safety testing.

These approaches primarily focus on external interfaces, particularly the CAN bus. Our work extends the fuzzing scope to include both external (CAN) and internal (SPI) communication buses.

General-Purpose MCU Fuzzing. Fuzz testing, developed for general-purpose MCU firmware, often relies on techniques that are not readily applicable to automotive ECUs. For example, μAFL [38] employs ARM's Embedded Trace Macrocell for detailed instruction tracing. While effective for certain MCUs, it requires specialized hardware features and debugging tools, which are costly and not typically available in ECUs. GDBFuzz [17] uses hardware breakpoints to capture execution states without code instrumentation. However, it can interfere with the real-time operation of ECUs, making it unsuitable for systems with strict timing requirements. SHIFT [40] introduces a semi-hosted fuzz testing framework that runs instrumented firmware on the MCU to collect coverage information. Although it provides detailed feedback, it necessitates code instrumentation, which can be impractical for ECUs due to limited memory resources and cost constraints.

Structure-Aware Fuzzing. Structure-aware fuzzing frameworks (e.g., Nautilus [2], Superion [62] and AFLSmart [50]) rely on code instrumentation, which is impractical for ECU firmware fuzzing due to limited resources and restricted access to firmware source code [67]. Several works [20, 39, 69] use QEMU-based emulation to collect coverage, but QEMU's incomplete support for automotive-specific architectures like TriCore makes them unsuitable for ECU firmware fuzzing. Protocol fuzzing tools like Peach [16] and BooFuzz [49] are not optimized for ECU firmware fuzzing and hence require non-trivial adaptation, and have shown comparable performance to automotive fuzzers [48].

6 Conclusion

We have proposed EcuFuzz, a structure-aware and diagnosis-guided fuzzing framework for automotive ECU firmware. Our evaluation has shown the compatibility of EcuFuzz with different ECU hardware architectures and the effectiveness of EcuFuzz in detecting faults in real-world ECUs.

7 Data Availability

All the experimental data and source code of our work is available at our replication site [15].

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No. 62372114 and 62332005).

References

[1] Harald Altinger, Franz Wotawa, and Markus Schurius. 2014. Testing methods used in the automotive industry: Results from a survey. In *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*. 1–6.

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars.. In *Proceedings of 2019 Network and Distributed System Security Symposium*.

[3] SIG Automotive. 2010. Automotive SPICE®. *Prozess Assessment Model* 2 (2010), 5.

[4] AUTOSAR. 2020. *E2E Protocol Specification*. Protocol Specification 30. AUTOSAR.

[5] AUTOSAR. 2022. AUTOSAR Website. https://www.autosar.org/ Accessed: 2024-10-31.

[6] AUTOSAR. 2022. *Specification of Operating System*. Software Specification 34. AUTOSAR. https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_SWS_OS.pdf

[7] AUTOSAR. 2022. Specification of SPI Handler/Driver. https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_SWS_SPIHandlerDriver.pdf Accessed: 2024-10-31.

[8] Enea Bagalini and Massimo Violante. 2016. Development of an automated test system for ECU software validation: An industrial experience. In *Proceedings of the 2016 15th Biennial Baltic Electronics Conference*. 103–106.

[9] Erik Bengtson. 2023. cantools: A Python package for handling CAN bus data. https://github.com/eerimoq/cantools Version 36.5.0.

[10] Richard Clubb et al. 2020. python-uds: An extensible UDS library for Python. https://github.com/richClubb/python-uds Latest commit: 95f3a53 (as of May 12, 2020).

[11] Wikipedia contributors. 2024. System basis chip. https://en.wikipedia.org/wiki/System_basis_chip Accessed: 2024-10-31.

[12] JA Cook and JS Freudenberg. 2007. Controller area network (can). *EECS* 461 (2007), 1–5.

[13] Jürgen Dobaj, Georg Macher, Damjan Ekert, Andreas Riel, and Richard Messnarz. 2023. Towards a security-driven automotive development lifecycle. *Journal of Software: Evolution and Process* 35, 8 (2023), e2407.

[14] Murray Dunne and Sebastian Fischmeister. 2022. Powertrace-based Fuzzing of CAN Connected Hardware. In *Proceedings of the 2022 IEEE International Conference on Cyber Security and Resilience*. 239–244.

[15] ECUFuzz. 2025. *ECUFuzz-2025*. https://github.com/ECUFuzz/ECUFuzz

[16] Michael Eddington. 2008. *Peach Fuzzer*. https://peachtech.gitlab.io/peach-fuzzer-community/

[17] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. 2023. Fuzzing embedded systems using debug interfaces. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1031–1042.

[18] Embien Technologies. 2024. Exploring the Major Electronic Control Units (ECUs) in Vehicle Systems. https://www.embien.com/automotive-insights/major-electronic-control-units-ecus-in-vehicle-systems Accessed: 2024-10-31.

[19] Krisztian Enisz, Denes Fodor, Istvan Szalay, and Laszlo Kovacs. 2014. Reconfigurable real-time hardware-in-the-loop environment for automotive electronic control unit testing and verification. *IEEE Instrumentation & Measurement Magazine* 17, 4 (2014), 31–36.

[20] Bo Feng, Alejandro Mera, and Long Lu. 2020. {P2IM}: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of 29th USENIX Security Symposium*. 1237–1254.

[21] International Organization for Standardization. 2021. *ISO/SAE 21434: Road vehicles — Cybersecurity engineering*. Technical Report ISO/SAE 21434:2021. ISO. https://www.iso.org/standard/70918.html

[22] Ford Motor Company. 2024. *Part 573 Safety Recall Report 24V-267*. Safety Recall Report 24V-267. National Highway Traffic Safety Administration. https://static.nhtsa.gov/odi/rcl/2024/RCLRPT-24V267-6161.PDF OMB Control No.: 2127-0004.

[23] Daniel S Fowler, Jeremy Bryans, Siraj Ahmed Shaikh, and Paul Wooderson. 2018. Fuzz testing for automotive cybersecurity. In *Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*. 239–246.

[24] Orhan Gazi, A Çağrı Arlı, Orhan Gazi, and A Çağrı Arlı. 2021. Serial Peripheral Interface. *State Machines using VHDL: FPGA Implementation of Serial Communication and Display Protocols* (2021), 143–192.

[25] PEAK-System Technik GmbH. 2024. PEAK-System Technik. https://www.peak-system.com/Products.57.0.html Accessed: 2024-10-31.

[26] Vector Informatik GmbH. 2024. Development and Test Tools for Automotive HIL and SIL Projects. https://www.vector.com/int/en/products/products-a-z/software/canoe/ Accessed: 2024-10-31.

[27] Oscar Haris, Sigit Wicaksono, Bambang Kurniawan, Gea Edytia, and Agus Darmawan. 2020. Design & analysis of external airbag system at the Toyota Venza vehicle. In *Proceedings of the 2020 6th International Conference on Computing Engineering and Design*. 1–5.

[28] Andreas Himmler, Klaus Lamberg, and Michael Beine. 2012. *Hardware-in-the-Loop Testing in the Context of ISO 26262.* Technical Report 2012-01-0035. SAE Technical Paper.

[29] Ikenna Chinazaekpere Ijeh. 2020. A collision-avoidance system for an electric vehicle: a drive-by-wire technology initiative. *SN Applied Sciences* 2, 4 (2020), 744.

[30] K Indu and M Aswatha Kumar. 2023. Electric vehicle control and driving safety systems: A review. *IETE Journal of Research* 69, 1 (2023), 482–498.

[31] International Organization for Standardization. 2011. *ISO 26262: Road Vehicles - Functional Safety.* Standard 26262. ISO.

[32] International Organization for Standardization. 2020. *Road vehicles – Unified diagnostic services (UDS).* Standard 14229. ISO. https://cdn.standards.iteh.ai/samples/72439/d6db7450800d4ccf859284e1a57bb23d/ISO-14229-1-2020.pdf

[33] Rolf Isermann, Jochen Schaffnit, and Stefan Sinsel. 1999. Hardware-in-the-loop simulation for the design and testing of engine-control systems. *Control Engineering Practice* 7, 5 (1999), 643–653.

[34] Parag Kharche, Meera Murali, and Geetanjali Khot. 2018. Uds implementation for ecu i/o testing. In *Proceedings of the 2018 3rd IEEE International Conference on Intelligent Transportation Engineering.* 137–140.

[35] Hyunghoon Kim, Yeonseon Jeong, Wonsuk Choi, Doon Hoon Lee, and Hyo Jin Jo. 2022. Efficient ECU analysis technology through structure-aware CAN fuzzing. *IEEE Access* 10 (2022), 23259–23271.

[36] Philip Koopman. 2014. A Case Study of Toyota Unintended Acceleration and Software Safety. https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf Accessed: 2024-10-31.

[37] Younho Lee, YangNam Lim, KokCheng Gui, Jin Seo Park, Pawan Reddy, and Syed Arshad Kazmi. 2015. *The study of AUTOSAR communication for automotive requirement.* Technical Report 2015-01-0185. SAE Technical Paper.

[38] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. 2022. μAFL: non-intrusive feedback-driven fuzzing for microcontroller firmware. In *Proceedings of the 44th International Conference on Software Engineering.* 1–12.

[39] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. 2021. DICE: Automatic emulation of DMA input channels for dynamic firmware analysis. In *Proceedings of 2021 IEEE Symposium on Security and Privacy.* 1938–1954.

[40] Alejandro Mera, Changming Liu, Ruimin Sun, Engin Kirda, and Long Lu. 2024. SHiFT: Semi-hosted Fuzz Testing for Embedded Applications. In *Proceedings of the 33th USENIX Security Symposium.* 5323–5340.

[41] Charlie Miller. 2015. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA* (2015), 1–91.

[42] Inc. Mitsubishi Motors North America. 2022. Inappropriate CVT ECU Software Programming – Safety Recall Campaign. https://static.nhtsa.gov/odi/rcl/2022/RCRIT-22V563-4075.pdf Accessed: 2024-10-31.

[43] Alexandros Mouzakitis, David Copp, Richard Parker, and Keith Burnham. 2009. Hardware-in-the-loop system for testing automotive ECU diagnostic software. *Measurement and control* 42, 8 (2009), 238–245.

[44] Sen Nie, Ling Liu, and Yuefeng Du. 2017. Free-fall: Hacking tesla from wireless to can bus. *Briefing, Black Hat USA* 25, 1 (2017), 16.

[45] Sen Nie, Ling Liu, Yuefeng Du, and Wenkai Zhang. 2018. Over-the-air: How we remotely compromised the gateway, BCM, and autopilot ECUs of Tesla cars. *Briefing, Black Hat USA* 91 (2018), 1–19.

[46] Tugsmandakh Nyamdelger, Munkhdelgerekh Batzorig, Esam Ali Albhelil, Yeji Koh, and Kangbin Yim. 2023. Fuzz testing and safe framework development for vehicle security analysis. In *Proceedings of the International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing.* 103–111.

[47] ODERA OHAZURIKE, CHISOM ONYENAGUBO, and CHRYSOGONUS OGOMAKA. 2024. Integrated Control Systems in Modern Automobiles. *Iconic Research And Engineering Journals* 7, 11 (2024), 126–132.

[48] Pranav Patki, Ajey Gotkhindikar, and Sunil Mane. 2018. Intelligent fuzz testing framework for finding hidden vulnerabilities in automotive environment. In *Proceedings of the 2018 Fourth International Conference on Computing Communication Control and Automation.* 1–4.

[49] Joshua Pereyda. 2015. *boofuzz: Network Protocol Fuzzing for Humans.* https://github.com/jtpereyda/boofuzz

[50] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1980–1997.

[51] Jiayi Qiang, Yong Gu, and Guochu Chen. 2020. FPGA Implementation of SPI bus communication based on state machine method. *Journal of Physics: Conference Series* 1449, 1 (2020), 012027.

[52] Ltd. Qingdao Kingst Electronics Co. 2020. Kingst Virtual Instruments User Guide. https://download.kamami.pl/p580653-Kingst_Virtual_Instruments_User_Guide.pdf Accessed: 2024-10-31.

[53] Andreea-Ina Radu and Flavio D Garcia. 2020. Grey-box analysis and fuzzing of automotive electronic components via control-flow graph extraction. In *Proceedings of the 4th ACM Computer Science in Cars Symposium.* 1–11.

[54] Md Abdur Rahim, Md Arafatur Rahman, Md Mustafizur Rahman, A Taufiq Asyhari, Md Zakirul Alam Bhuiyan, and D Ramasamy. 2021. Evolution of IoT-enabled connectivity and applications in automotive industry: A review. *Vehicular Communications* 27 (2021), 100285.

[55] Matteo Sonza Reorda and Massimo Violante. 2006. Hardware-in-the-loop-based dependability analysis of automotive systems. In *Proceedings of the 12th IEEE International On-Line Testing Symposium.* 6–pp.

[56] Memoona Sadaf, Zafar Iqbal, Abdul Rehman Javed, Irum Saba, Moez Krichen, Sajid Majeed, and Arooj Raza. 2023. Connected and automated vehicles: Infrastructure, applications, security, critical challenges, and future aspects. *Technologies* 11, 5 (2023), 117.

[57] Muhammad Salman Sarfraz, Hyunsoo Hong, and Seong Su Kim. 2021. Recent developments in the manufacturing technologies of composite components and their cost-effectiveness in the automotive industry: A review study. *Composite Structures* 266 (2021), 113864.

[58] Brian Thorne et al. 2024. python-can: The Controller Area Network (CAN) package for Python. https://github.com/hardbyte/python-can Latest commit: a39e63e (as of April 14, 2024).

[59] Luc van Dijk. 2017. Future Vehicle Networks and ECUs: Architecture and Technology Considerations. https://www.nxp.com/docs/en/white-paper/FVNECUA4WP.pdf Accessed: 2024-10-31.

[60] Manu Jo Varghese, Adnan Anwar, Frank Jiang, and Robin Doss. 2024. Novel CAN Bus Fuzzing Framework for Finding Vulnerabilities in Automotive Systems. In *Proceedings of the 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume*. 56–58.

[61] Manu Jo Varghese, Frank Jiang, Robin Doss, Adnan Anwar, and Abdur Rakib. 2024. Adaptive Fuzz Testing for Automotive ECUs: A Modular Testbed Approach for Enhanced Vulnerability Detection. In *Proceedings of the ACM SIGCOMM 2024 Conference: Posters and Demos*. 110–112.

[62] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *Proceedings of 2019 IEEE/ACM 41st International Conference on Software Engineering*. 724–735.

[63] Timothy Werquin, Mathijs Hubrechtsen, Ashok Samraj Thangarajan, Frank Piessens, and Jan Tobias Muehlberg. 2019. Automated Fuzzing of Automotive Control Units. In *Proceedings of the 2019 International Workshop on Secure Internet of Things*. 1–8.

[64] Samuel Woo, Hyo Jin Jo, and Dong Hoon Lee. 2014. A practical wireless attack on the connected car and security protocol for in-vehicle CAN. *IEEE Transactions on intelligent transportation systems* 16, 2 (2014), 993–1006.

[65] Anthony Kee Teck Yeo, Matheus E Garbelini, Sudipta Chattopadhyay, and Jianying Zhou. 2023. VitroBench: manipulating in-vehicle networks and COTS ECUs on your bench: a comprehensive test platform for automotive cybersecurity research. *Vehicular Communications* 43 (2023), 100649.

[66] Clinton Young, Joseph Zambreno, Habeeb Olufowobi, and Gedare Bloom. 2019. Survey of automotive controller area network intrusion detection systems. *IEEE Design & Test* 36, 6 (2019), 48–55.

[67] Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. 2022. Fuzzing of embedded systems: A survey. *Comput. Surveys* 55, 7 (2022), 1–33.

[68] Haichun Zhang, Kelin Huang, Jie Wang, and Zhenglin Liu. 2021. Can-ft: A fuzz testing method for automotive controller area network bus. In *Proceedings of the 2021 International Conference on Computer Information Science and Artificial Intelligence*. 225–231.

[69] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic firmware emulation through invalidity-guided knowledge inference. In *Proceedings of 30th USENIX Security Symposium*. 2007–2024.

Received 2025-02-18; accepted 2025-03-31