Preprints.org

Article

Not peer-reviewed version

Algorithmic Optimization for Accelerated UDS Fuzzing in Cyber-Physical Automotive Networks: The BB-FAST Approach on LIN-Bus

Sungsik Im *, Yijoon Jung, Junyoung Park

Posted Date: 27 February 2026

doi: 10.20944/preprints202602.1228.v1

Keywords: cyber-physical networks; automotive security; fuzz testing; optimized fuzzing algorithm; unified diagnostic services (UDS); local interconnect network (LIN)

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Algorithmic Optimization for Accelerated UDS Fuzzing in Cyber-Physical Automotive Networks: The BB-FAST Approach on LIN-Bus

Sungsik Im *, Yijoon Jung and Junyoung Park

Security Vulnerability Analysis Team, FESCARO Co., Ltd., Suwon-si, Gyeonggi-do 16512, Republic of Korea
* Correspondence: sungsik.im@fescaro.com; Tel.: +82-10-9737-2709

Abstract

In modern cyber-physical vehicle networks, the security of component-level Electronic Control Units (ECUs) is essential for overall system reliability. While CAN bus security is well-studied, the Local Interconnect Network (LIN) has received less attention despite its growing role in critical functions and diagnostic services (UDS). The inherent constraints of the LIN protocol, specifically its low bandwidth and Master-Slave architecture, make traditional fuzz testing impractical due to extremely long execution times. This paper proposes BB-FAST, an optimized framework for faster vulnerability detection in LIN-based systems. By integrating batch processing and binary search techniques, BB-FAST overcomes communication bottlenecks and enables efficient error localization. Experiments on a physical automotive ECU show that BB-FAST significantly reduces testing time—by 55.56% and 93.44% depending on the diagnostic session—ensuring high efficiency even under frequent reset conditions. By mitigating these physical limitations through algorithmic optimization, this work enables thorough security verification for LIN-based diagnostic interfaces that was previously constrained by protocol latency, thereby enhancing the integrity of cyber-physical automotive networks.

Keywords: cyber-physical networks; automotive security; fuzz testing; optimized fuzzing algorithm; unified diagnostic services (UDS); local interconnect network (LIN)

---

1. Introduction

With the rapid transition of the automotive industry toward software-defined vehicles (SDV) [1], modern vehicles have evolved into complex cyber-physical networks (CPNs) where the physical movement of the vehicle is governed by intricate electronic control units (ECUs). As advanced driver assistance systems (ADAS) and infotainment systems become more sophisticated, the functional significance of these ECUs has increased, making the reliability and security of component-level nodes critical factors in determining the overall safety and resilience of the vehicular ecosystem[2].

In these cyber-physical environments, a single vulnerable ECU can serve as an entry point to compromise the entire network's functional integrity. While traditional automotive security focused on external interfaces—such as Wi-Fi, cellular, and Bluetooth—modern cybersecurity emphasizes the necessity of securing internal networks, including the Controller Area Network (CAN), Automotive Ethernet, and even low-speed protocols. Accordingly, international standards like ISO/SAE 21434 [3] and UN Regulation No. 155 (UNR 155) [4] now require rigorous security testing to ensure the trust and safety of internal communication architectures.

A key element in evaluating ECU security is the testing of Unified Diagnostic Services (UDS) [5]. Since UDS provides powerful control functions such as memory management and firmware updates, it is a primary target for adversaries seeking to exploit vehicular control systems. To proactively identify vulnerabilities in these diagnostic interfaces, fuzz testing—which injects malformed inputs to verify exception-handling capabilities—is widely utilized. While UDS-based fuzzing via CAN is a

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

well-established requirement [6], research on other critical internal protocols has been comparatively limited.

Concurrently, the Local Interconnect Network (LIN) is an indispensable protocol used in various domains, including side mirror control, intelligent HVAC sensors, and Battery Management Systems (BMS) [7]. In next-generation architectures, LIN-based ECUs are increasingly undertaking more complex control functions, leading to a significant rise in the integration of UDS for advanced lifecycle management and maintenance. However, despite this growing importance, UDS-based fuzzing for LIN remains under-explored. Specifically, LIN is constrained by a limited bandwidth of up to 20 kbps and a rigid Master-Slave architecture. These physical communication constraints lead to severe performance bottlenecks, making conventional testing methodologies designed for CAN environments impractical for LIN-based cyber-physical nodes.

To bridge this gap, this paper proposes BB-FAST, an optimized UDS-based fuzzing framework specifically designed to overcome the inherent constraints of LIN communication. By introducing algorithmic optimizations such as batch processing and binary-search logic, our approach addresses the physical latency of LIN with cyber intelligence. This ensures deterministic efficiency and facilitates exhaustive security verification, enabling thorough vulnerability detection in bandwidth-limited environments that were previously difficult to test rigorously.

The remainder of this paper is organized as follows. Section 2 provides the theoretical background on LIN message structures and fuzzing techniques, followed by a review of related research. Section 3 details the proposed UDS-based fuzzing method-ologies, optimized for the LIN environment through preliminary analysis. Section 4 presents a theoretical analysis and complexity modeling for each proposed method. Section 5 evaluates the effectiveness of the proposed approach through experimental validation on a representative LIN-based ECU. Section 6 discusses the significance of the findings and addresses the study's limitations. Finally, Section 7 concludes the paper with a summary and future directions.

2. Background and Related Works

2.1. Background

2.1.1. Local Interconnect Network (LIN)

The local interconnect network (LIN) is a low-cost communication protocol developed to complement the cost limitations of the CAN, which handles high-speed communication within in-vehicle networks. LIN is primarily utilized in environments that do not require high bandwidth, such as for controlling seats, doors, and side mirrors, as well as for communication between various sensors. Currently, LIN is defined as an international standard through the ISO 17987 series and plays an essential role in vehicle networks [8–10].

In the case of CAN, it supports communication speeds of up to 1 Mbps and features a multi-master approach where multiple nodes can communicate simultaneously. Furthermore, it demonstrates high reliability due to structures such as Cyclic Redundancy Check (CRC) error detection and differential signaling (CAN High/Low). Conversely, LIN supports a relatively slow communication speed of up to 20 kbps—approximately 1/50th the speed of CAN—and is characterized by a single-master architecture where one master node controls all slave nodes [11]. Additionally, LIN features a simple structure using a single wire, resulting in lower message reliability compared to CAN but offering the advantage of being more cost-effective.

A LIN message frame, as shown in Figure 1, consists of a header frame transmitted by the master node and a response frame transmitted by either the master or a slave node depending on the specific identifier. The header frame comprises a break field, a sync field, and a protected identifier (PID) field, which respectively serve as a frame start notification, synchronization, and message ID value. Notably, PIDs ranging from 0x00 to 0x3B are used for signal transmission, while 0x3C is utilized for requests in diagnostic functions and 0x3D for responses. The response frame has a data field of up to 8 bytes and includes a 1-byte Checksum field for data integrity. Since LIN is based on the universal

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

asynchronous receiver-transmitter (UART) communication method, every field constituting a LIN frame includes a start bit and a stop bit. That is, a field with a size of 1 byte (8 bits) consists of a total of 10 bits, including the start and stop bits.

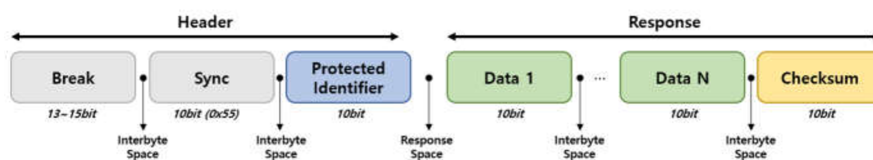

Figure 1. Frame of the LIN Message.

Unlike CAN, where all nodes act as master nodes, LIN has a structure with only one master node and multiple slave nodes associated with it. In this architecture, message transmission is conducted via a polling method where the master node transmits a header frame to the bus according to a schedule table to call a specific slave node. In particular, for diagnostic communication such as UDS as illustrated in Figure 2, the master node transmits a request header with a PID of 0x3C and the corresponding data to the bus. Subsequently, when the master node transmits a response header with a PID of 0x3D along with an empty data frame, the slave node corresponding to the node address (NAD) receives the 0x3D message, populates it with the response data, and transmits the response message to the bus. While this method prevents message collisions and ensures stable communication, it acts as a constraint that degrades overall latency and the efficiency of security testing, as the slave node must wait until it receives a polling header from the master node to respond.

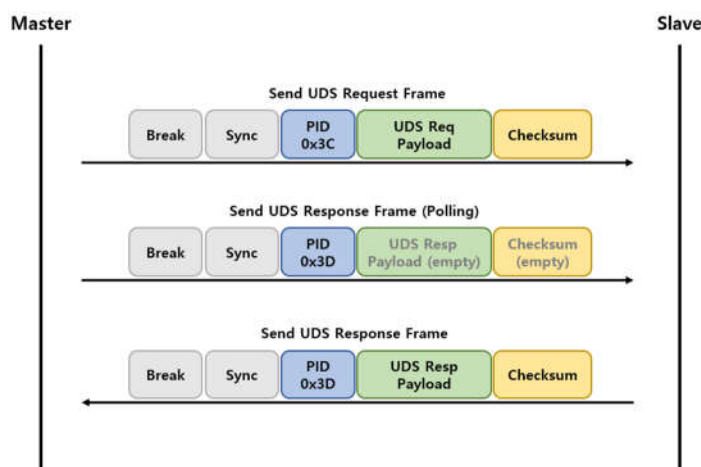

Figure 2. Example of LIN Communication on UDS.

2.1.2. Unified Diagnostic Services (UDS)

Unified diagnostic services (UDS) is an automotive diagnostic communication protocol defined by the ISO 14229 standard [12,13], which supports standardized diagnostic communication at the application layer. This service facilitates standardized communication between in-vehicle ECUs and external diagnostic clients. It plays a critical role in vehicle service and maintenance by performing core functions such as controller configuration, memory dumps, and firmware updates.

The ISO 14229 standard assigns a unique service identifier (SID) to each service for vehicle diagnostics, and its primary functions are categorized as shown in Table 1. UDS operates on a request-response pair mechanism; when a client transmits a request frame containing a specific SID, the ECU returns either a positive response or a negative response frame. In this process, the SID of a positive response is the value of the request SID plus 0x40. Conversely, the SID of a negative response is 0x7F, which is returned along with a negative response code (NRC) that provides specific information regarding the nature of the response.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026
doi:10.20944/preprints202602.1228.v1

Table 1. Examples of Service ID on UDS Standard.

Request SID	Response SID	Service	Details
0x10	0x50	Diagnostic Session Control	Control which UDS services are available
0x11	0x51	ECU Reset	Reset the ECU
0x27	0x67	Security Access	Enable use of security-critical services via authentication
0x22	0x62	Read Data By Identifier	Read data from targeted ECU
0x23	0x63	Read Memory By Address	Read data from physical memory
0x2E	0x6E	Write Data By Identifier	Program specific variables determined by data parameters
0x3D	0x7D	Write Memory By Address	Write information to the ECU's memory
0x31	0x71	Routine Control	Initiate/stop routines
0x34	0x74	Request Download	Start request to add software/data to ECU
0x35	0x75	Request Upload	Start request to read software/data from ECU
0x36	0x76	Transfer Data	Perform actual transfer of data
	0x7F	Negative Response	Sent with a negative response code when a request cannot handled

While the UDS protocol provides a consistent application layer interface, its implementation is tailored to the characteristics of the underlying physical and data link layers. Specifically, Part 7 of the ISO 14229 standard [14] specifies the implementation of UDS over the LIN communication environment. To identify individual slave nodes on the LIN bus, a NAD value is utilized; this value is defined within the payload of the designated diagnostic PIDs, 0x3C(Request) and 0x3D(Response). As illustrated in Figure 3, a LIN diagnostic frame—which has a maximum data field of 8 bytes—consists of a 1-byte NAD, a 1-byte Protocol Control Information (PCI) field defining the frame type and data length, and a 1-byte SID. These are followed by up to 5 bytes of actual service data or parameters.

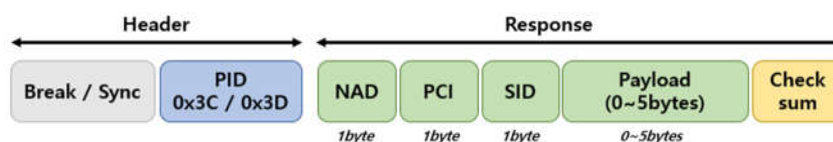

Figure 3. Example UDS frame on LIN.

2.1.3. Fuzz Testing

Fuzz testing (fuzzing) is a dynamic analysis technique used to identify software defects and security vulnerabilities by continuously injecting random or abnormal data into a target system. Fuzzing is highly effective even in black-box environments where source code is unavailable, as it can trigger exception-handling logic or unexpected system crashes. This allows researchers to detect unknown vulnerabilities and evaluate software stability within undefined or undocumented regions.

Fuzz testing is primarily classified into two categories based on the methodology used to generate test cases:

- Mutation-based Fuzzing: This approach generates modified data by applying techniques such as bit-flipping or random value injection to existing normal data packets, which serve as seed

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

samples. It can be implemented without an in-depth understanding of complex protocol structures and is mainly utilized for testing simple network packets.

- Generation-based Fuzzing: This method creates new test data from scratch that complies with the rules of the target protocol, such as frame structures and checksums, based on its specification. This approach is particularly effective for testing sophisticated communication protocols and diagnostic services, such as UDS.

2.2. Related Works

2.2.1. Trends in Automotive Network Security Surveys

In-vehicle network (IVN) security has evolved significantly with the transition toward software-defined architectures. Rathore et al. [15] categorized in-vehicle communication architectures and primary internal protocols such as CAN, LIN, Flex-Ray, and Automotive Ethernet. They surveyed security solutions focused on machine learning-based Intrusion Detection Systems (IDS) and encryption-based authentication mechanisms. However, the majority of the presented security solutions and intrusion detection research were conducted on CAN-based internal networks, with limited coverage of security solution cases for LIN-based communication.

Luo et al. [16] provided a comprehensive classification of automotive cybersecurity testing methodologies, suggesting that penetration testing, vulnerability scanning, and fuzzing-based security verification are core technologies for vehicle security validation. Notably, this study analyzed that attack research targeting internal vehicle networks accounts for the largest proportion of automotive security literature. They reported numerous studies on fuzzing and penetration testing for major protocols like CAN, Scalable Service-Oriented Middleware over IP (SOME/IP), Diagnostics over IP (DoIP), and UDS. Conversely, security testing and fuzzing research for LIN were notably absent from the literature distribution. Beyond these surveys, several comprehensive studies have highlighted that while cyber-physical networks require holistic protection, research efforts remain disproportionately focused on high-speed protocols like CAN and Automotive Ethernet [17,18].

2.2.2. Fuzzing and Vulnerability Detection for In-vehicle Networks

Automated fuzzing is a primary method for detecting implementation flaws in ECUs. I. Cho et al. [19] implemented a tool to scan sessions and services to understand ECU environments for security testing within UDS frameworks and conducted tests targeting the CAN environment of specific ECUs. This process effectively identified undocumented custom sessions and services within the ECUs. D. Kim et al. [20] analyzed the potential for detecting vulnerabilities in ECU internal software based on the CAN protocol, with the test subjects and fuzzing input structures designed according to the CAN frame format. However, their study did not include fuzzing experiments that considered the frame structure or the Master-Slave scheduling characteristics of the LIN protocol.

In parallel, recent research has explored intelligent fuzzing and mutation-based techniques to improve the detection rate of unknown vulnerabilities across various in-vehicle networks [21,22]. However, these frameworks often rely on the high bandwidth and multi-master capabilities of CAN and Automotive Ethernet, which allow for rapid feedback loops and high-concurrency testing. Such approaches are difficult to translate directly to the LIN protocol, where the rigid Master-Slave architecture and low baud rates impose significant physical communication constraints. Consequently, existing fuzzing methodologies frequently fail to account for the deterministic scheduling and inherent latency of LIN, necessitating a specialized optimization approach that considers the unique physical-layer characteristics of the bus.

2.2.3. Efficiency and Optimization in Security Testing

The efficiency of security testing is a critical factor in cyber-physical systems where resource constraints are significant. General research in the field of software testing has long utilized batch-processing and adaptive search algorithms to optimize search space exploration [23–25]. In the

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

automotive domain, some attempts have been made to reduce testing time through structural-aware fuzzing or predictive models [21]. Nevertheless, systematic optimization of the fuzzing process for low-bandwidth protocols like LIN—specifically addressing the bottleneck of Master-Slave scheduling and recovery time—remains a largely unexplored area. This research gap necessitates a dedicated research and framework, which optimizes communication overhead while ensuring the resilience of the testing process.

3. Proposed Methodology

This chapter introduces the proposed system and fuzzing methodologies for performing UDS-based fuzz tests in a LIN environment. As illustrated in Figure 4, the proposed system operates in three primary stages: scanning the UDS sessions and services present in the target ECU, identifying valid services, and executing fuzz tests for each identified service.

The UDS session and service scanning processes build upon the research of I. Cho et al. [19], but have been advanced through specialized steps tailored for the LIN environment and an optimized logic that eliminates redundant operations upon encountering negative responses. From the perspective of component-level ECU testing within a LIN environment, the normal operation of the controller during the fuzz testing is verified by checking for a valid response to the UDS request message. Furthermore, this paper proposes three distinct methodologies for verifying the operational integrity and normal behavior of the ECU during the fuzzing process.

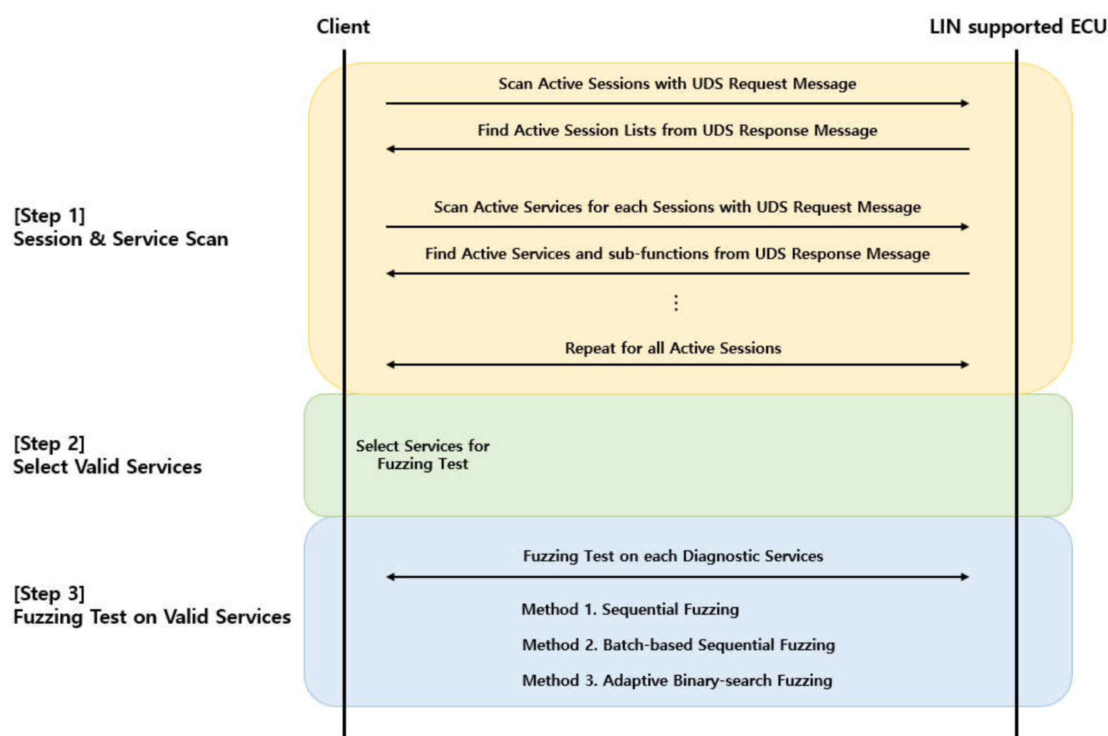

Figure 4. Overview of the proposed fuzz testing system.

3.1. Session & Service Scan

To perform fuzzing in a UDS environment, the client scans all active diagnostic sessions on the target ECU and explores a session transition diagram based on the breadth-first search (BFS) algorithm. Upon identifying all sessions, the proposed methodology navigates through each session according to the session transition diagram to enumerate all active services and their sub-functions.

The active sessions on the target ECU are determined through positive or negative responses to Diagnostic Session Control (0x10) service requests. Figure 5 illustrates the message flow between the client and the target ECU for session scanning. The client transmits a Diagnostic Session Control (0x10)

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

request message using PID 0x3C to scan for active sessions and generates a polling message to the ECU node to receive the response.

In response to the request, the target ECU populates the received polling message frame with the appropriate PCI, SID, payload, and checksum values. In this study, the responses returned by the target ECU for session scanning are categorized into the following three cases:

- Case 1: If a positive response with SID 0x50 is returned for a SID 0x10 request, it is determined that the target session is accessible from the current session.
- Case 2: If a negative response with NRC 0x7F 0x12 (Sub-function Not Supported) is returned for a SID 0x10 request, it is determined that the target session is inaccessible from the current session.
- Case 3: If a negative response with an NRC value other than 0x12 is returned for a SID 0x10 request, the client directly determines the accessibility of the target session based on the specific NRC received.

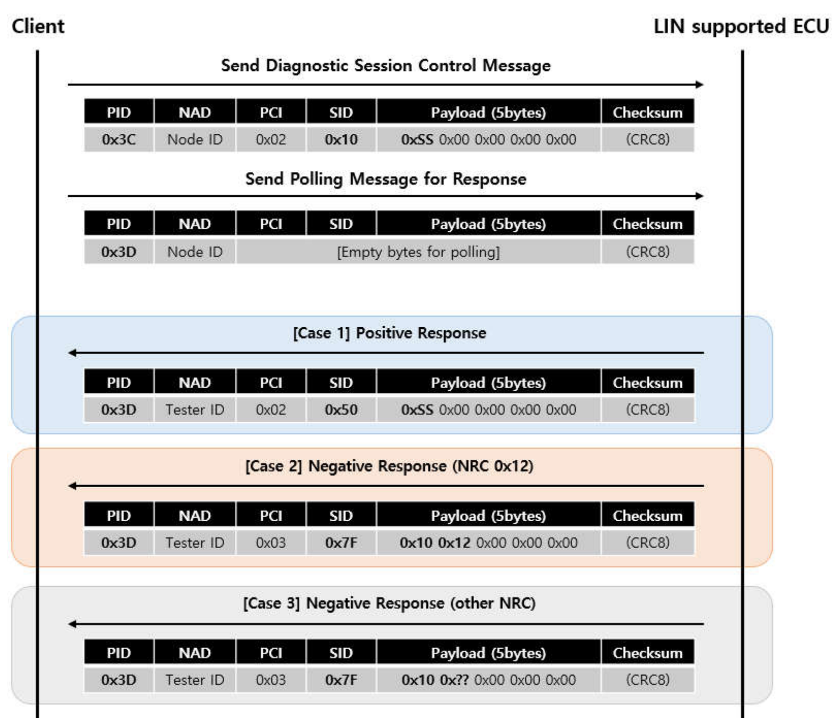

Figure 5. Flow graph of the scanning session step.

Figure 6 illustrates the algorithm for discovering all sessions within the ECU. Starting from the default session (0x01), the algorithm identifies all accessible sessions from the current state and sequentially explores them based on the BFS algorithm.

Subsequently, after accessing each session based on the transition diagram explored during the session scanning process, the client scans for the list of active services and the available sub-function values for each service. Figure 7 shows the message flow for scanning services and sub-functions within each session. The client transmits a request message for the diagnostic service to be scanned using PID 0x3C and delivers a polling message to the ECU node to receive a response. For services that support sub-functions, their availability is verified by sequentially incrementing the sub-function value from 0x00 within a predefined range. Additionally, the Tester Present (0x3E) service is invoked to maintain the non-default session state, preventing an automatic reversion to the default session due to timeout during the scanning process.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Algorithm 1 BFS-based Diagnostic Session Scan

Require: Default session s_0, candidate session set $\mathcal{S}$, NAD, retry limit K
Ensure: Shortest transition paths $paths[s]$, transition edges E

```
1:  paths ← {s0 : [s0]}
2:  E ← ∅
3:  Q ← queue initialized with s0
4:  while Q is not empty do
5:      u ← pop(Q)
6:      Pu ← paths[u]
7:      SendUDS(NAD, 0x10, s0)                    ▷ reset to default session
8:      for all x ∈ Pu do
9:          ok ← false
10:         for i = 1 to K do
11:             resp ← SendUDS(NAD, 0x10, x)
12:             if resp == POS(0x50, x) then
13:                 ok ← true ; break
14:             end if
15:         end for
16:         if ok == false then
17:             continue while-loop
18:         end if
19:     end for
20:     for all v ∈ S do
21:         if v ∈ paths or v == u then
22:             continue
23:         end if
24:         ok ← false
25:         for i = 1 to K do
26:             resp ← SendUDS(NAD, 0x10, v)
27:             if resp == POS(0x50, v) then
28:                 ok ← true ; break
29:             end if
30:         end for
31:         if ok == true then
32:             E ← E ∪ (u, v)
33:             paths[v] ← Pu ⊕ [v]
34:             push(Q, v)
35:         end if
36:         SendUDS(NAD, 0x10, s0)
37:             x ∈ Pu
38:             SendUDS(NAD, 0x10, x)
39:         end for
40:
41:     end while
        return paths, E
```

Figure 6. Algorithm of session scan.

The target ECU responds to the service request by populating the received polling message frame with the appropriate PCI, SID, payload, and checksum values. In this study, the response cases that the target ECU can return during service scanning are classified into the following three categories:

- Case 1: If a positive response (0xSV + 0x40) is returned for a specific service (0xSV) request, it is determined that the target service exists within the current session.
- Case 2: If a negative response with NRC 0x7F 0x11 (Service Not Supported) is returned for a specific service (0xSV) request, it is concluded that the target service does not exist in the current session. Consequently, the process bypasses any subsequent sub-function discovery for that service and immediately proceeds to the next service ID to optimize scanning efficiency.
- Case 3: If a negative response with an NRC value other than 0x11 is returned for a specific service (0xSV) request, the client directly determines the existence of the target service based on the received NRC.

Figure 8 illustrates the algorithm for traversing all services within the ECU. Service and sub-function scans are performed sequentially across all active sessions identified during the session scanning phase.

3.2. Select Valid Services

Based on the sub-function information of available sessions and services identified through the scanning process, the target UDS services for fuzzing are selected. This stage can be conducted using distinct selection methodologies depending on the requirements of various stakeholders—such as the testing entity, ECU manufacturer, or cybersecurity audit manager—as well as compliance with diverse certification regulations.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

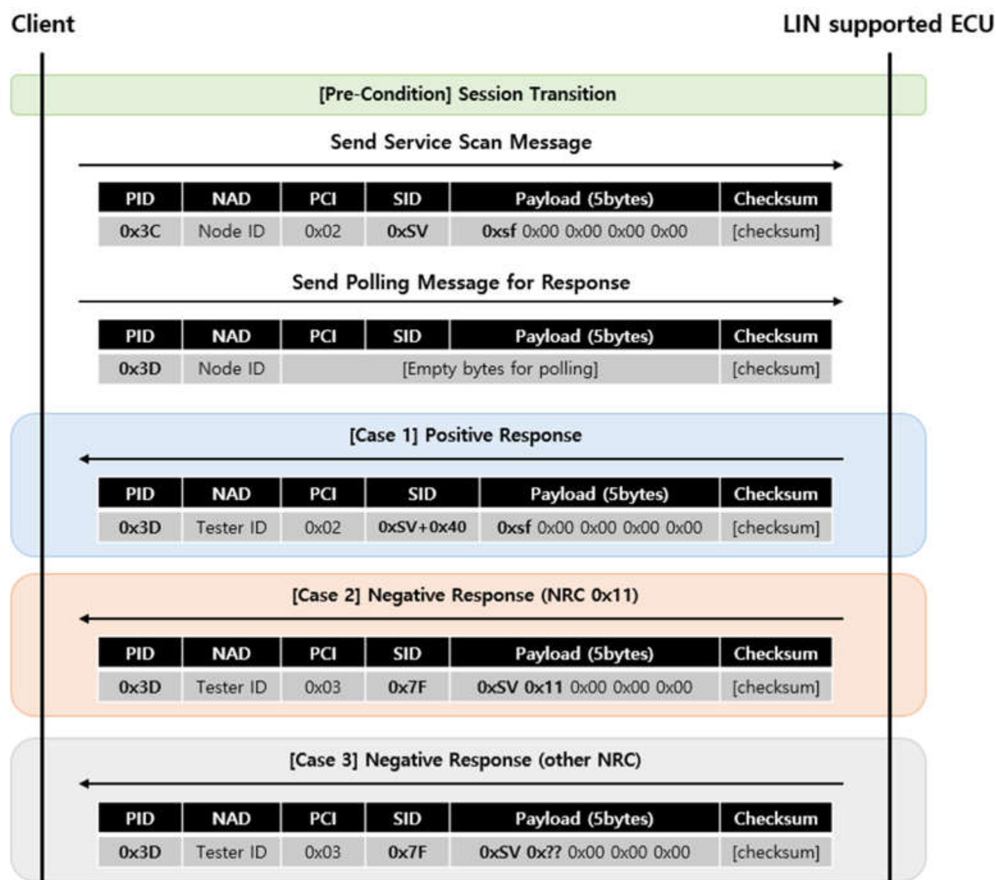

Figure 7. Flow graph of the scanning service step.

Algorithm 2 Path-aware UDS Service Scan with NRC=0x11 Shortcut

Require: Discovered paths $paths[s]$, service ID range $\mathcal{D}$, sub-function set $\mathcal{F}$, NAD

Ensure: Result table $T(session, path, SID, SUB, code)$

1: **for all** $s \in paths$ **sorted by** path length **do**
2: $P \leftarrow paths[s]$
3: $\text{SendUDS}(NAD, 0x10, 0x01)$ ▷ reset to default session
4: **for all** $x \in P$ **do**
5: $resp \leftarrow \text{SendUDS}(NAD, 0x10, x)$
6: **if** $resp \neq \text{POS}(0x50, x)$ **then**
7: **continue next session**
8: **end if**
9: **end for**
10: **for all** $SID \in \mathcal{D}$ **do**
11: $resp_0 \leftarrow \text{SendUDS}(NAD, SID, 0x00)$
12: **if** $resp_0 == \text{NEG}(0x7F, SID, 0x11)$ **then**
13: **for all** $SUB \in \mathcal{F}$ **do**
14: $\text{Append}(T, (s, P, SID, SUB, \text{“11”}))$
15: **end for**
16: **continue next SID**
17: **end if**
18: **for all** $SUB \in \mathcal{F}$ **do**
19: $resp \leftarrow \text{SendUDS}(NAD, SID, SUB)$
20: $code \leftarrow \text{Interpret}(resp, SID)$
21: $\text{Append}(T, (s, P, SID, SUB, code))$
22: **end for**
23: **end for**
24: **end for**
return T

Figure 8. Algorithm of service scan.

As presented in Table 2, the proposed methodology prioritizes active services that support sub-functions or additional parameters as primary fuzzing targets. Specifically, it is proposed that fuzzing be mandatory for core functions related to security authentication, data retrieval, and operational control, such as Security Access (0x27), Read Data by Identifier (0x22), and Routine Control (0x31).

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026
doi:10.20944/preprints202602.1228.v1

Furthermore, undocumented custom services and services without additional parameters may also be included within the scope of the test.

Table 2. Recommended service to select for fuzz testing.

Request SID	Service	Selection
0x10	DiagnosticSessionControl	◎
0x11	EcuReset	○
0x19	ReadDTCInformation	○
0x22	ReadDataByIdentifier	◎
0x23	ReadMemoryByAddress	◎
0x27	SecurityAccess	◎
0x28	CommunicationControl	○
0x2C	DynamicallyDefineDataIdentifier	◎
0x2E	WriteDataByIdentifier	◎
0x31	RoutineControl	◎
0x34	RequestDownload	◎
0x35	RequestUpload	◎
0x36	TransferData	◎
0x3D	WriteMemoryByAddress	◎
0x3E	TesterPresent	○
0x83	AccessTimingParameters	○
0x85	ControlDTCSetting	○
0x86	ResponseonEvent	○
0x87	LinkControl	○
0x10	DiagnosticSessionControl	○

◎: Mandatory, ○: Recommend.

3.3. Methods for Fuzz Testing

Fuzzing is conducted based on the previously scanned session and service lists to detect vulnerabilities—such as improper exception handling for abnormal inputs and unexpected system crashes—and to evaluate stability within undefined regions of the target ECU. Furthermore, this study aims to perform UDS-based fuzzing to verify the security of component-level ECUs within a LIN network.

Considering the specific characteristics of LIN, such as its Master-Slave architecture—where slave nodes can only respond through the master node's polling messages—and its relatively low transmission speed, this paper proposes three UDS-based fuzzing methodologies categorized by the approach used to verify malfunctions and anomalies. Each proposed method transmits separate UDS request messages to monitor for normal processing and valid responses, thereby identifying potential ECU malfunctions.

3.3.1. Sequential Discovery Fuzzing (SDF)

The Sequential Discovery Fuzzing (SDF) test conducts fuzzing on a per-service basis and verifies the normal operation status of the ECU by transmitting a polling message for every individual fuzzing request. While this approach is similar to standard fuzzing procedures where responses are monitored for all test cases to determine malfunctions, the inherent characteristics of LIN

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

communication require the master node to transmit an individual polling message to retrieve each response. The detailed operational algorithm for this method is illustrated in Figure 9.

Algorithm 3 Sequential Discovery Fuzzing (SDF)
Input: TotalMessages N
Output: ErrorList

```
1: for i = 1 to N do
2:     send_fuzz_message(i)
3:     response ← send_polling_message()   ▷ Polling (e.g., TesterPresent 0x3E)
4:     if response = NO_RESPONSE ∨ response = ERROR then
5:         ErrorList.add(i)
6:         reset_ecu_power()               ▷ Hard Reset via Relay or Power Supply
7:         re-enter_session(0x10)          ▷ Diagnostic Session Control
8:     end if
9: end for
```

Figure 9. Algorithm of SDF Test.

3.3.2. Batch-Based Sequential Fuzzing (BSF)

The Batch-based Sequential Fuzzing(BSF) test executes the fuzzing process by dividing the total test range(N) into batches of a specific size(m). Unlike the previously proposed SDF test, the BSF method just transmits all fuzz messages within a defined batch consecutively. To verify the operational integrity and processing status of the ECU after batch transmission, the system sends a session transition (re-entry) request to the current diagnostic session followed by a polling message, determining the state based on the response from the target node.

If a positive response is received, the test proceeds to the next batch and repeats the process. If a negative response is returned, the system performs an SDF procedure within that specific batch range to identify the exact error-inducing message. Once the fault is pinpointed, the BSF process resumes from the subsequent message with a full batch size, continuing the cycle. The detailed operational algorithm for this methodology is illustrated in Figure 10.

Algorithm 4 Batch-based Sequential Fuzzing (BSF)
Require: TotalMessages N, BatchSize m
Ensure: ErrorList

```
 1: i ← 1
 2: while i ≤ N do
 3:     upper_limit ← min(i + m − 1, N)
 4:                                         ▷ 1. Batch Transmission
 5:     for j = i to upper_limit do
 6:         send_fuzz_message(j)
 7:     end for
 8:                                         ▷ 2. Verification
 9:     status ← check_session_and_polling()
10:     if status = SUCCESS then
11:         i ← upper_limit + 1
12:     else
13:         reset_ecu_power()
14:         re-enter_session(0x10)
15:         for k = i to upper_limit do
16:             send_fuzz_message(k)
17:             if check_polling() = FAIL then
18:                 ErrorList.add(k)
19:                 reset_ecu_power()
20:                 re-enter_session(0x10)
21:                 i ← k + 1
22:                 break
23:             end if
24:         end for
25:     end if
26: end while
```

Figure 10. Algorithm of BSF Test.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

doi:10.20944/preprints202602.1228.v1

3.3.3. Batch-based Binary-Search Fuzzing and Accelerated Security Testing (BB-FAST)

Similar to the BSF test, the Batch-based Binary-search Fuzzing and Accelerated Security Testing (BB-FAST) test divides the total test range(N) into batches of a specific size(m). It transmits all fuzz messages within a batch and then evaluates the operational integrity and processing status of the ECU by sending a session transition (re-entry) request to the current session followed by a polling message.

For a positive response, the process proceeds to the subsequent batch. In contrast to the BSF test, if a negative response is returned, the BB-FAST identifies the error-triggering message through a binary search. Within the batch range where the error occurred, the algorithm explores the messages using a binary search mechanism; at each step of this search, the system transmits a session transition (re-entry) request and a polling message to verify the response of the target node.

Upon identifying the message that induced the error, the BB-FAST process resumes from the next message with a full batch size, repeating the sequence. The detailed operational algorithm for this methodology is illustrated in Figure 11.

Algorithm 5 Batch-based Binary-search Fuzzing (BB-FAST)

Input: TotalMessages N, BatchSize m
Output: ErrorList

```
1: function FindErrorBinary(low, high)
2:     if low = high then
3:         return low
4:     end if
5:     mid ← ⌊(low + high)/2⌋
6:                                                        ▷ Test left half
7:     for j = low to mid do
8:         send_fuzz_message(j)
9:     end for
10:    status ← check_session_and_polling()
11:    if status = FAIL then
12:        reset_ecu_power()
13:        re-enter_session(0x10)
14:        return FindErrorBinary(low, mid)        ▷ Narrow down to left
15:    else                        ▷ Left is normal, error is in the right half
16:        return FindErrorBinary(mid + 1, high)
17:    end if
18: end function

19: i ← 1
20: while i ≤ N do
21:    upper_limit ← min(i + m − 1, N)
22:    for j = i to upper_limit do
23:        send_fuzz_message(j)
24:    end for
25:    status ← check_session_and_polling()
26:    if status = SUCCESS then
27:        i ← upper_limit + 1
28:    else
29:        reset_ecu_power()
30:        re-enter_session(0x10)
31:        error_idx ← FindErrorBinary(i, upper_limit)
32:        ErrorList.add(error_idx)
33:        reset_ecu_power()
34:        re-enter_session(0x10)
35:        i ← error_idx + 1
36:    end if
37: end while
```

Figure 11. Algorithm of BB-FAST.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

4. Theoretical Analysis on Proposed Methods

This chapter evaluates the three proposed fuzzing methodologies introduced in Section 3 by calculating their respective time complexities and providing a comparative analysis. Variables that significantly impact the performance of the fuzzing process—including the total number of messages, batch size, and the number of identified vulnerabilities or errors are defined. By establishing these time complexity models, we analyze the most effective fuzzing approach across different experimental conditions and scenarios.

4.1. Time-Complexity Analysis

To calculate the time complexity for each of the three proposed fuzzing methodologies, the variables that impact performance are defined in Table 3.

Table 3. Variable symbols with time-complexity.

Symbol	Description	Unit
N	Total number of fuzzing messages (packets)	
m	Batch size	
k	Number of errors (vulnerabilities) in the target controller $(k \ll N)$	
T_f	Transmission time for each fuzzing message	ms
T_p	Session Re-entry and Polling Message Verification Time	ms
T_r	ECU initialization time (power re-application, boot-up, etc.)	ms

For SDF, polling process is performed for each of the N fuzzing packets to verify the response. Furthermore, for every error identified in the target ECU, additional time is required for a power cycle (power cycling the ECU to perform a hard reset). The time complexity of SDF testing is expressed as Equation (1):

$$T_{SDF} = N(T_f + T_p) + k \cdot T_r \tag{1}$$

For BSF, fuzzing messages are sent for all N packets, while polling messages are transmitted for every batch of size m to verify the response. For each detected error, the ECU undergoes one power cycle, followed by an SDF procedure within the batch range to pinpoint the specific error. Assuming that errors are uniformly distributed within the batch of size m, the average number of attempts required to identify a single error is defined as $\frac{m+1}{2}$, based on the expected value of a discrete uniform distribution. For the purpose of complexity analysis, this expected value is simplified to $\frac{m}{2}$. The total time complexity of BSF testing is expressed as Equation (2):

$$T_{BSF} = \left(N \cdot T_f + \frac{N}{m} T_p\right) + k\left(2T_r + \frac{m}{2}(T_f + T_p)\right) \tag{2}$$

BB-FAST operates similarly to BSF, but it utilizes a binary search within the batch range of size m for each detected error to identify the specific error-inducing message. The total time complexity of BB-FAST is expressed as Equation (3):

$$T_{BBF} = \left(N \cdot T_f + \frac{N}{m} T_p\right) + k\left(2T_r + \sum_{i=1}^{\log_2 m}\left(\frac{m}{2^i} T_f + T_p + T_r\right)\right) \tag{3}$$

The calculated time complexities are represented using Big-O notation for worst-case scenarios in Table 4, considering the parameters N, m, and k, which determine the system scale and status. The analysis results demonstrate that the proposed batch-based techniques (BSF and BB-FAST) achieve superior efficiency compared to SDF by drastically reducing the frequency of response verifications from the total number of packets(N) to the batch unit($\frac{N}{m}$). In particular, BB-FAST maintains high stability and minimizes performance degradation, even in environments with

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

frequent communication delays or ECU resets, by reducing the recovery complexity from linear (m) to logarithmic ($\log m$).

Table 4. Comparison of theoretical time complexity for the proposed fuzzing methods.

Method	Time-complexity (Big-O)
SDF	$O(N+k)$
BSF	$O(\frac{N}{m}+km)$
BB-FAST	$O(\frac{N}{m}+k\log m)$

Notes: N=total test cases, k=number of errors, m=batch size.

4.2. Performance Evaluation via Numerical Simulation

In this section, numerical simulations are performed under various environments to compare and evaluate the practical efficiency of the three proposed fuzzing methodologies. These simulations reflect the actual constraints of the LIN bus environment, such as the potential number of errors (k), the practical ECU recovery time (T_r), and the polling cost (T_p) for verifying normal operation. The variables used in the simulations follow the definitions in Table 3, and the common parameters are established in Table 5.

Table 5. Fixed Parameters for numerical simulation.

Symbol	Default Value	Notes
N	65,536	2-byte range (0x0000~0xFFFF)
T_f	10ms	Frame transmission time at LIN 20kbps
T_p	100ms	0x10 Service polling latency included

4.2.1. Simulation According to Error Density (k)

This subsection analyzes the performance of each fuzzing methodology based on the number of vulnerabilities present in the target ECU. The parameters established for this analysis are as follows:

- k: 1 to 100 (independent variable)
- m: 512 (fixed value)
- T_r: 2.0 sec (fixed value; reflects the human reaction limit for controllability as specified in ISO 26262-3 [26])

Figure 12 illustrates the simulated test duration as a function of the number of errors, the independent variable. As the number of errors increases, BB-FAST demonstrates a gentler rate of time increase compared to BSF due to its search efficiency. This indicates that even when numerous vulnerabilities are detected in a large-scale test case environment, the time required to pinpoint the error locations remains controlled below a certain threshold, thereby ensuring overall test availability.

4.2.2. Analysis According to ECU Recovery Time (T_r)

This subsection evaluates the performance of the fuzzing methodologies according to the time required for ECU recovery after an error occurs. The parameters for this analysis are:

- k: 10 (fixed value)
- m: 512 (fixed value)
- T_r: 0.5 sec to 5.0 sec (independent variable)

Figure 13 presents the simulated test duration as a function of the ECU recovery time. The results indicate that with a batch size of 512, a recovery time of approximately 3 seconds serves as the crossover point where the time efficiency of BB-FAST surpasses that of BSF. However, in environments where the batch size is set larger—considering scenarios where N exceeds 2 bytes (e.g.,

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

3 bytes)—BB-FAST is expected to maintain its performance advantage through its efficient search structure, even in high-performance ECU environments with significant recovery times.

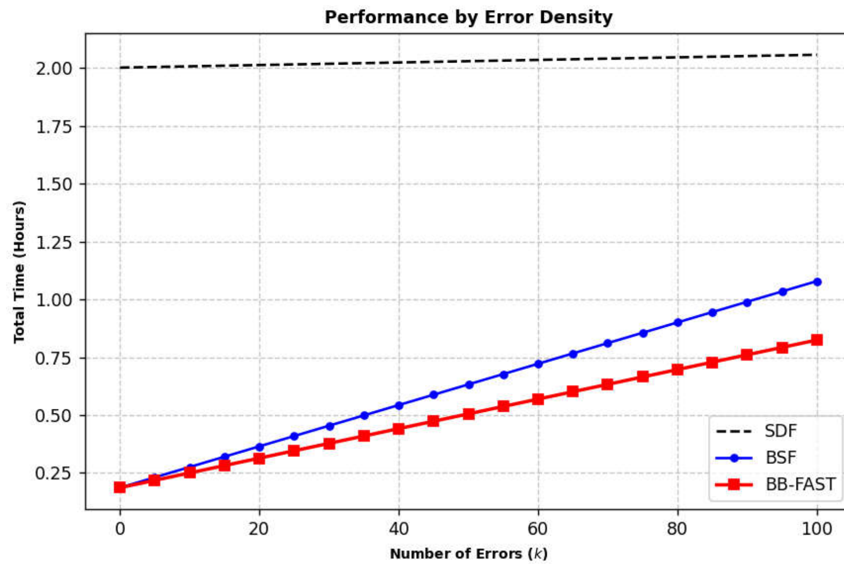

Figure 12. Simulation results according to error density.

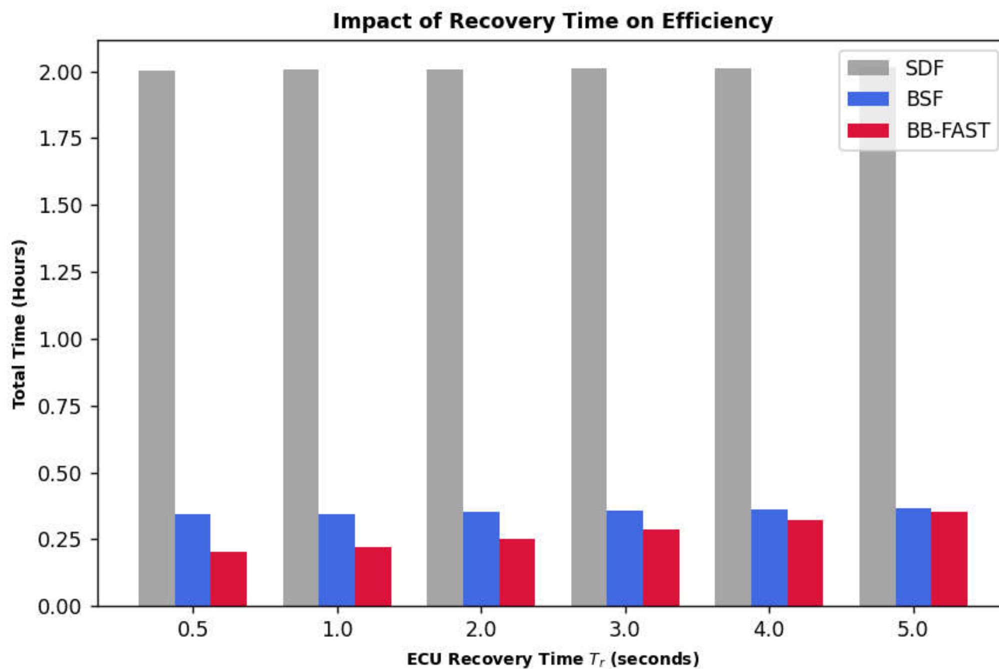

Figure 13. Simulation results according to ECU recovery time.

4.2.3. Simulation According to Batch Size (m)

This subsection analyzes the performance of each fuzzing methodology relative to the batch size. The parameters for this analysis are:

- k: 10 (fixed value)
- m: 10 to 1,000 (independent variable)
- T_r: 2.0 s (fixed value; reflects the human reaction limit for controllability as specified in ISO 26262-3 [26])

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

Figure 14 shows the simulated test duration between BSF and BB-FAST based on the batch size. An intersection in performance between BSF and BB-FAST occurs at a batch size of approximately 300. Notably, from a batch size of approximately 100, the test duration for BSF begins to increase at a much steeper gradient compared to BB-FAST.

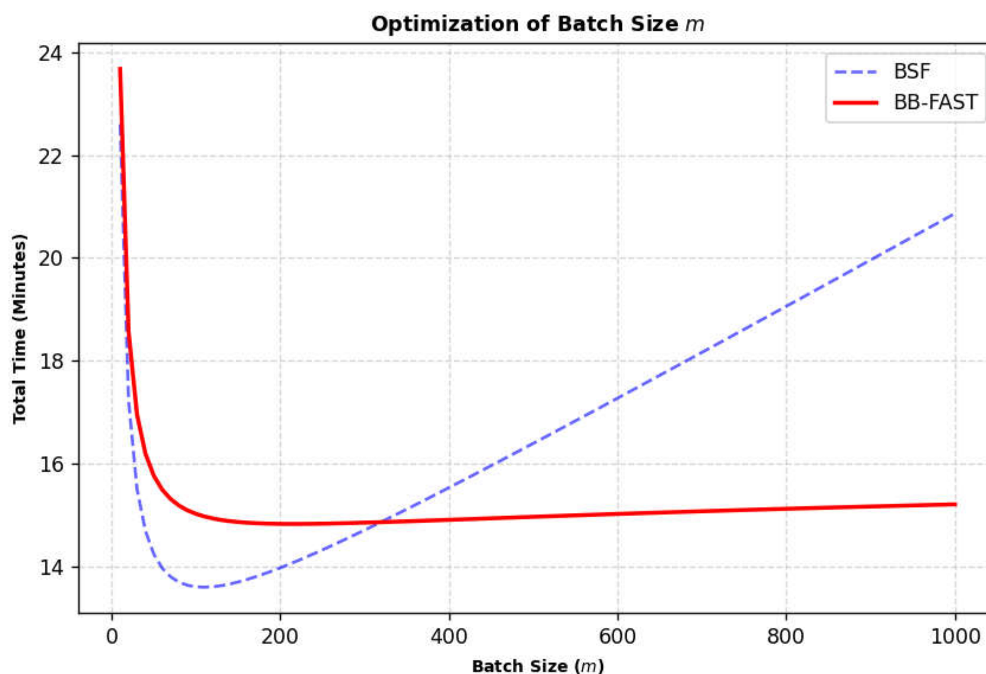

Figure 14. Simulation results according to batch size.

4.2.4. Summary of Simulation Results

The results of the numerical simulations conducted in this section confirm that the proposed BB-FAST methodology demonstrates the highest scalability in environments with limited bandwidth and significant reset overhead, such as the LIN bus. The key findings and academic implications derived from the simulations are as follows:

- Strategic Selection of Parameter m: As confirmed in Section 4.2.3, the theoretical optimal batch size varies depending on error density (k) and recovery time (T_r). However, this paper adopts m = 512 as a strategic standard, comprehensively considering the scalability for future tests (e.g., 3-byte search spaces), the minimization of polling overhead, and the implementation efficiency of the binary search algorithm. This design ensures universal performance across diverse vehicle network environments rather than seeking a local optimum for a specific scenario.
- Ensuring Feasibility for Large-scale Search Spaces: The simulation results show that an m = 512 configuration suppresses the frequency of polling—the dominant bottleneck in total execution time—to 1/512 of the original rate. This minimizes the time delay caused by polling overhead (T_p) even in vast search spaces exceeding 2 bytes, thereby guaranteeing deterministic throughput to complete fuzz tests within restricted project schedules.
- Optimization of Search Efficiency Under Physical Constraints: Even in environments with high reset overhead (e.g., T_r=2.0 s), BB-FAST maintains efficient performance in locating error points through binary search. While the linear search cost of BSF increases sharply as the batch size grows, BB-FAST exhibits a gentle logarithmic increase, demonstrating robustness against system instability (presence of multiple errors) or large-scale batch environments.

In conclusion, considering the characteristics of LIN communication and the physical constraints of actual vehicle controllers, BB-FAST-based testing is the most stable and efficient approach for performing UDS-based fuzzing.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

5. Experimental Results

In this section, we evaluate the proposed system by performing fuzz tests on an actual automotive ECU equipped with LIN-based UDS. We developed specific tools to implement the three proposed fuzzing methodologies and conducted experiments on the target controller within a standardized experimental environment.

5.1. Experimental Setup

The hardware and software environments for the UDS-based fuzz tests on the LIN bus are configured as detailed in Tables 6 and 7. Based on the system mechanism proposed in Section 3, we developed a testing tool and executed it on the physical ECU.

Table 6. Hardware configuration and specifications.

Component	Specification	Role
Laptop	Intel(R) Core(TM) Ultra 7, 32GB RAM	developing tool, controlling fuzz testing scripts
PEAK PLIN-USB	supports all LIN specifications	USB to LIN bus converter
Automotive ECU	LIN-based Electronic Control Unit	experimental target
Power Relay	12V/5A, 1-Channel Relay	ECU power relay and reset

Table 7. Software and development environment.

Component	Specification	Role
Operating System	Windows 11 Pro	
Language	Python 3.13.9	script developing, test controlling
Module	PLinApi	LIN communication API
	numpy, matplotlib	data processing, simulation and result plotting
Monitoring	PLIN-View	real-time LIN monitoring

5.2. Experiment Process

The overall experimental procedure follows the methodology described in Section 3. First, we identify the available services for each session in the target ECU through a session and service scan. Subsequently, a specific service is selected to perform a comparative performance experiment among the three proposed fuzzing techniques.

The experiment assumes a black-box environment where internal structures and vulnerability information of the target ECU are unknown. The total execution time is measured as the primary metric for evaluating the efficiency of each methodology. Detailed parameters are configured identically to the values in Section 4.2.1. Specifically, in the event of an error requiring an ECU power cycle, the recovery time (T_r) from the moment of the error to the resumption of the test is fixed at 2 seconds.

In this experiment, Read Data by Identifier(SID 0x22) was selected from the identified sessions, and the performance of the three fuzzing techniques was verified by transmitting 65,536 payloads (equivalent to a 2-byte range) for that service.

5.3. Experiment Results

This subsection analyzes and compares the results of the three proposed fuzzing methodologies (SDF, BSF, and BB-FAST) based on the established experimental environment and procedures.

5.3.1. Measurement of Execution Time and Analysis of Results

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

The results of scanning the sessions and services in the target ECU and performing fuzzing on a 2-byte data field for a selected service are summarized in Table 8. With experimental ECU, a total of three sessions and seven services were identified. During the experiment, the target ECU responded normally to all requests without any anomalies, such as abnormal terminations or communication delays.

Table 8. Fuzz testing measurement results.

Session	Service ID	Execution Time (SDF)	Execution Time (BSF)	Execution Time (BB-FAST)
0x01	0x22	121 min 48sec	7min 53sec	7min 51sec
0x02	0x22	17 min 56sec	7min 50sec	7min 49sec
0x03	0x22	121 min 48sec	7min 53sec	7min 51sec

The experimental results show that in Session 0x01 and Session 0x03, the batch-based algorithms (BSF and BB-FAST) achieved a time reduction of approximately 93.44% compared to the traditional sequential approach (SDF). In Session 0x02, a time reduction of approximately 55.56% was observed. This efficiency gain is attributed to the minimization of network idle time and physical response latency through batch processing. Notably, Session 0x02, being a programming session, demonstrated a faster processing speed compared to other sessions due to its specialized response handling.

5.3.2. Analysis of Performance in Error-Free Scenarios

In this specific test, the execution times for BSF and BB-FAST were nearly identical. This is because, in the "Best Case" scenario where no errors occur, BB-FAST executes the same batch verification logic as BSF. Although the binary search-based error identification logic—the core feature of BB-FAST—was not activated, the results confirm that BB-FAST maintains the same high-performance levels as BSF under normal operational conditions.

5.3.3. Comparison and Discussion with Theoretical Analysis

The similarity in execution times for BSF and BB-FAST in this experiment can be further explained through the time complexity analysis conducted in Section 4.2. According to the theoretical analysis, the complexity of both algorithms converges to $\Omega\left(\frac{N}{m}\right)$ in the best-case scenario where no errors are detected. The fact that both techniques achieved a significant reduction in execution time compared to SDF proves that the theoretical efficiency of batch processing remains valid in actual controller environments. Although the binary search logic of BB-FAST was not triggered due to the absence of ECU errors (corresponding to k=0 as defined in Section 4.2.1), it is expected that BB-FAST would outperform BSF in environments where errors occur, as demonstrated in our previous complexity analysis.

6. Discussions and Limitations

The experimental results validate that **BB-FAST** effectively overcomes the inherent performance bottlenecks of the **LIN** protocol, which have long hindered comprehensive security testing. As demonstrated in Section 5.3, the proposed framework achieved significant time reductions—**55.56% and 93.44%** depending on the session—compared to traditional **SDF**. These results confirm that the theoretical model ($\Omega\left(\frac{N}{m}\right)$) established in Section 4.2 is highly consistent with real-world controller behavior. Notably, even in the best-case scenario where no errors occur (k=0), BB-FAST maintains performance equivalent to the optimized BSF without introducing additional computational or communication overhead.

From a practical perspective, the primary significance of this study lies in its ability to close the security gap in low-bandwidth in-vehicle networks. Due to the extremely slow baud rate of LIN (up

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

to 20 kbps), performing exhaustive UDS-based fuzzing has often been considered impractical, leading to a tendency to either simplify or entirely omit security verification for LIN-based ECUs. BB-FAST transforms this time-consuming burden into a feasible industrial process. By significantly reducing the testing duration and ensuring high test availability through its binary search logic—especially when dealing with the significant latency of power cycles (T_r) —our framework allows for rigorous and exhaustive security audits that were previously avoided due to time constraints.

Regarding the limitations of this study, while the theoretical analysis and simulations suggest consistent performance across various environments, the empirical experiments were limited to a single LIN-based controller and a 2-byte payload range. In complex vehicular architectures, recovery times (T_r) and response behaviors may vary significantly across different ECU types. Additionally, high-bandwidth protocols like Automotive Ethernet or data-heavy systems like In-Vehicle Infotainment (IVI) present different challenges in terms of data volume and protocol complexity. Therefore, future research should focus on verifying the scalability of BB-FAST across diverse hardware architectures and advancing toward multivariate fuzzing algorithms with expanded payload ranges to detect more complex, unknown vulnerabilities.

7. Conclusions

This study proposed BB-FAST, an optimized framework designed to address the critical time constraints and test availability issues in LIN-based cybersecurity testing. Based on the modeling of the controller fuzzing environment and verification through empirical experiments, the final conclusions are as follows:

- Solving the LIN Performance Bottleneck: By restructuring fuzzing requests into batch units, BB-FAST achieved a substantial reduction in execution time—specifically by 55.56% and 93.44%—proving that algorithmic optimization can effectively mitigate the physical limitations of low-bandwidth protocols.
- Ensuring Resilience in Error Scenarios: Through the integration of binary search logic, the framework maintains high efficiency under normal conditions while effectively suppressing time delays during error-induced resets to a logarithmic scale, ensuring a robust and uninterrupted testing process.
- Expanding the Scope of Practical Security Testing: This research provides a scalable solution to include LIN-based components, which were previously difficult to test rigorously, into the standard security verification suite. By maximizing test feasibility, BB-FAST ensures that core diagnostic protocols in modern cyber-physical automotive systems are thoroughly verified against potential threats.

While the results of this study were derived from specific controller and protocol environments, the underlying logic of the proposed algorithms is extensible to various automotive communication architectures. Future research will focus on integrating intelligent fuzzing logic to enhance detection rates for unknown vulnerabilities and ensuring scalability toward high-speed networks, such as Automotive Ethernet. We expect the findings of this study to serve as foundational data for establishing standardized methodologies for automotive cybersecurity verification.

Author Contributions: Conceptualization, S.I., I.J. and J.P.; methodology, S.I., I.J. and J.P.; software, I.J.; validation, I.J. and J.P.; formal analysis, S.I.; investigation, S.I. and I.J.; resources, S.I.; data curation, I.J.; writing—original draft preparation, S.I.; writing—review and editing, S.I., I.J. and J.P.; visualization, S.I. and I.J.; supervision, J.P.; project administration, J.P.; funding acquisition, J.P. All authors have read and agreed to the published version of the manuscript.

Funding: Please add: This research and the APC was funded by the Korea Evaluation Institute of Industrial Technology (KEIT), grant funded by the Korea government (MOTIR) (No. 2410013725).

Institutional Review Board Statement: Not applicable.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026
doi:10.20944/preprints202602.1228.v1

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on reasonable request from the corresponding author due to corporate security policies and the sensitivity of the vehicle control system data.

Acknowledgments: The authors would like to thank D. Kim, S. Jeon and J. Jang from the Security Vulnerability Analysis Team at FESCARO for their technical support. We also extend our gratitude to J. Kim and E. Ju from the IP Strategy Team at FESCARO for their administrative assistance.

Conflicts of Interest: The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. De, M.; Pesé, M.D.; Brooks, R.R.; Hasan, M. Contextualizing Security and Privacy of Software-Defined Vehicles: State of the Art and Industry Perspectives. *arXiv* **2024**, https://doi.org/10.48550/arXiv.2411.10612.
2. Anwar, A.; Anwar, A.; Moukahal, L.; Zulkernine, M. Security Assessment of In-Vehicle Communication Protocols. *Vehicular Communications* **2023**, 44, 100639, https://doi.org/10.1016/j.vehcom.2023.100639.
3. ISO/SAE 21434; Road Vehicles — Cybersecurity Engineering. ISO: Geneva, Switzerland, 2021.
4. UN Regulation No. 155; Cyber Security and Cyber Security Management Syste. UNECE: Geneva, Switzerland, 2021;
5. Yekta, A.R.; Loza, N.; Gramm, J.; Schneider, M.P.; Katzenbeisser, S. From ECU to VSOC: UDS Security Monitoring Strategies. The Nineteenth International Conference on Emerging Security Information, Systems and Technologies(SECURWARE 2025); IARIA: 26-30 October 2025; pp.40–47, https://doi.org/10.48550/arXiv.2510.25375.
6. Kayas, G.; Etas, Z.P.; Etas, D.G.; Etas, T.A.; Payton, J. AI-Assisted Vulnerability Analysis And Classification Framework for UDS on CAN-Bus Fuzzer. In Proceedings of the 10th escar USA - The World's Leading Automotive Cyber Security Conference; ESCRYPT: Michigan, USA, 2023.
7. Ruff, M. Evolution of Local Interconnect Network (LIN) Solutions. 2003 IEEE 58th Vehicular Technology Conference(VTC 2003-Fall); IEEE Cat: Florida, USA, 2003, pp.3382-3389, https://doi.org/10.1109/VETECF.2003.1286317.
8. ISO 17987-1; Road Vehicles — Local Interconnect Network (LIN) — Part 1: General Information and Use Case Definition. ISO: Geneva, Switzerland, 2025.
9. ISO 17987-2; Road Vehicles — Local Interconnect Network (LIN) — Part 2: Transport Protocol and Network Layer Services. ISO: Geneva, Switzerland, 2025.
10. ISO 17987-3; Road Vehicles — Local Interconnect Network (LIN) — Part 3: Protocol Specification. ISO: Geneva, Switzerland, 2025.
11. Navet, N.; Simonot-Lion, F. *Automotive Embedded Systems Handbook (Industrial Information Technology)*; 2008;
12. ISO 14229-1; Road Vehicles — Unified Diagnostic Services (UDS) — Part 1: Application Layer. ISO: Geneva, Switzerland, 2020.
13. ISO 14229-2; Road Vehicles — Unified Diagnostic Services (UDS) — Part 2: Session Layer Services. ISO: Geneva, Switzerland, 2021.
14. ISO 14229-7; Road Vehicles — Unified Diagnostic Services (UDS) — Part 7: UDS on Local Interconnect Network (UDSonLIN). ISO: Geneva, Switzerland, 2022.
15. Rathore, R.S.; Hewage, C.; Kaiwartya, O.; Lloret, J. In-Vehicle Communication Cyber Security: Challenges and Solutions. *Sensors* **2022**, 22, https://doi.org/10.3390/s22176679
16. Luo, F.; Zhang, X.; Yang, Z.; Jiang, Y.; Wang, J.; Wu, M.; Feng, W. Cybersecurity Testing for Automotive Domain: A Survey. *Sensors* **2022**, 22, https://doi.org/10.3390/s22239211
17. Hussain, I.; Reis, M.J.C.S.; Serôdio, C.; Branco, F. A Bibliometric Analysis and Visualization of In-Vehicle Communication Protocols. *Future internet* **2025**, 17, https://doi.org/10.3390/fi17060268
18. Yu, J.; Wagner, S; Wang, B.; Luo, F. A systematic mapping study on security countermeasures of in-vehicle communication systems. *SAE International Journal of Transportation Cybersecurity and Privacy* **2021**, 4, https://doi.org/10.48550/arXiv.2105.00183

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 27 February 2026 doi:10.20944/preprints202602.1228.v1

19. Cho, I.; Yoon, J.; Eom, S. Implementation of session and service scanner for UDS security testing. KASE 2023 Annual Spring Conference; 24-27 May 2023; pp.893-897.
20. Kim, H.; Jeong, Y.; Choi, W.; Lee, D.H.; Jo, H.J. Efficient ECU Analysis Technology Through Structure-Aware CAN Fuzzing. *IEEE Access* **2022**, 10, 23259–23271, https://doi.org/10.1109/access.2022.3151358.
21. Chen, Q.; Zikui, K.; Hu, K.; Peng, X.; Gong, S.; Chen, B.; Kong, Z.; Jiang, H.; Sun, B.; Lu, Y. Structure-Aware, Diagnosis-Guided ECU Firmware Fuzzing. *Proc. ACM Softw. Eng* **2025**, 2, pp.871–893, https://doi.org/10.1145/3728914.
22. Faschang, T.; Macher, G. An Open Software-Based Framework for Automotive Cybersecurity Testing. Communications in Computer and Information Science; Springer: Cham, Switzerland, 2023; 1890, pp.316–328, https://doi.org/10.1007/978-3-031-42307-9_22.
23. Zeller, A.; Hildebrandt, R. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* **2002**, 28, pp.183–200, https://doi.org/10.1109/32.988498.
24. Kapugama, C.G. Extending Delta Debugging Minimization for Spectrum-Based Fault Localization, 2026 IEEE 16th Annual Computing and Communication Workshop and Conference (CCWC); IEEE SE: Las Vegas, USA, 5-7, Jan 2026, https://doi.org/10.48550/arXiv.2601.04689.
25. McMinn, P. Search-Based Software Test Data Generation: A Survey. *STVR* **2004**, 14, pp.105–156, https://doi.org/10.1002/stvr.294.
26. ISO 26262-3; Road Vehicles — Functional Safety — Part 3: Concept Phase. ISO: Geneva, Switzerland, 2018.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.