2024-01-2799 Published 09 Apr 2024

Comparing Open-Source UDS Implementations Through Fuzz Testing

Levent Çelik Clemson University

John McShane Eastern Michigan University

Christian Scott, Iwinosa Aideyan, Richard Brooks, and Mert D. Pese Clemson University

Citation: Çelik, L., McShane, J., Scott, C., Aideyan, I. et al., "Comparing Open-Source UDS Implementations Through Fuzz Testing," SAE Technical Paper 2024-01-2799, 2024, doi:10.4271/2024-01-2799.

Received: 09 Nov 2023 Revised: 01 Feb 2024 Accepted: 05 Feb 2024

Abstract

In the ever-evolving landscape of automotive technology, the need for robust security measures and dependable vehicle performance has become paramount with connected vehicles and autonomous driving. The Unified Diagnostic Services (UDS) protocol is the diagnostic communication layer between various vehicle components which serves as a critical interface for vehicle servicing and for software updates. Fuzz testing is a dynamic software testing technique that involves the barrage of unexpected and invalid inputs to uncover vulnerabilities and erratic behavior. This paper presents the implementation of fuzz testing methodologies on the UDS layer, revealing the potential vulnerabilities that could be exploited by malicious entities.

By employing both open-source and commercial fuzzing tools and techniques, this paper simulates real-world scenarios to assess the UDS layer's resilience against anomalous data inputs. Specifically, we deploy several open-source UDS implementations on a Controller Area Network (CAN) testbed and use them as a target for the aforementioned fuzzing tools. The outcomes of the fuzzing campaigns provide both automakers and researchers with insights about the completeness of open-source UDS implementations, as well as existing vulnerabilities. Our recommendations are intended to inform researchers and developers about the current state of these implementations, especially if they consider integrating them into their products. Ultimately, the use of open-source implementations in the automotive domain promises a more secure, easier to maintain, safer, and cheaper development process.

This paper underscores the significance of continuous testing and fortification in ensuring the integrity of automotive systems with a particular focus on UDS, offering a valuable contribution to the advancement of secure vehicular technology.

Introduction

The adaptation of electronics by the automotive sector has evolved the current generation of vehicles into smart and connected machines. Modern vehicles can carry Internet and Bluetooth capabilities, electronic engine control and steering, a connected entertainment system, several cameras and screens, driving assist, and even autonomous driving. To provide such functionality, vehicles contain an internal network of Electronic Control Units (ECUs), which are small embedded devices with individual responsibilities. This internal network is commonly referred to as an In-Vehicle Network (IVN).

Several criteria attract customers to a vehicle. Comfort, ease of driving, fuel efficiency, emission rates, and perhaps most importantly safety are among what is expected from a modern vehicle. In contrast, perhaps because automobiles have been around much longer than electronics have and such a concern had never been necessary, the cybersecurity of a vehicle is often not taken into account by the average customer; or, it is assumed to be at a satisfactory level. This means that the manufacturers' investments into cybersecurity are effectively not marketable, thus not directly profitable. Moreover, vehicle security is researched and developed by manufacturers as proprietary software and is not publicly verifiable unless reverse engineered. As demonstrated by the many existing vulnerabilities in modern vehicles [1, 2, 3, 4, 5, 6], such individual efforts by manufacturers often fall short. Thus, we argue that good vehicle security should be the product of the collective efforts of research institutions and individual researchers; and, proprietary tools and software are not a good basis for security because they are less accessible and harder to verify. This work particularly focuses on the availability of UDS, the *de facto* application layer standard for in-vehicle diagnostics.

Currently, studying UDS is an expensive and unfeasible task. The UDS implementations used in vehicles are

proprietary and can only be accessed in a black-box setting. Moreover, different vehicle models often have different UDS implementations, even if they are manufactured by the same manufacturer. Thus, verifying the state of the art UDS security requires testing each model or ECU individually. Not only can these tests result in permanent loss of functionality on ECUs, it also requires the work of experts; thus, it comes with a non-negligible cost. On the other hand, proper open-source UDS implementations would allow a wider range of approaches to security, leading to a more rapid development and wider verification, while also being cost efficient and less prone to errors. Moreover, since the UDS standard gets updated periodically and its implementations require software patches, having a shared implementation would cut from such maintenance efforts. Lastly, a publicly verifiable UDS implementation could help eliminate security by obscurity in the automotive field. To see the current state of publicly available UDS implementations, this paper tests open-source UDS implementations by employing fuzz testing tools and techniques, as well as manually crafted tests.
Our Contributions:

- We survey and analyze the functionality of open-source UDS implementations;
- To the best of our knowledge, we are the first academic work to deploy a fuzzer on these implementations;
- Through fuzz testing and manual test inputs, we assess the UDS implementations' functionality and stability;
- We compare an open-source and a proprietary fuzzer for automotive penetration testing.

Background

In-Vehicle Network (IVN) Protocols

Several IVN protocols are used for different purposes inside a modern vehicle. CAN, LIN, FlexRay, MOST, and Ethernet are a notable few. We focus primarily on CAN [8], which is the predominant IVN protocol [6] owing to its cost-effectiveness and simplified manufacturing process. It allows for priority communication between ECUs using message IDs [6, 9], and has real time guarantees on message delivery times. It is a broadcast protocol without source or destination addresses. These factors ensure CAN's value for vehicles that require strict guarantees for message delivery times and prioritized messages; however, it was not designed with security in mind [2, 9], is unencrypted and unauthenticated, and it only supports a bandwidth of up to 1 Mbps.

An IVN protocol that has been recently receiving interest by researchers and OEMs alike is Automotive Ethernet due to its higher bandwidth compared to other IVNs, and the existing higher layer protocols built on it such as IP and TCP. Although it is simply referred as "Ethernet" in automotive context, Automotive Ethernet differs slightly from its regular counterpart. Mainly, Automotive Ethernet is designed to have higher resistance to signal noise, lower cost of wiring, and better electromagnetic immunity, which are all critical requirements for IVN networks. Current implementations support a bandwidth of up to 1 Gbps; and, higher data rates are being standardized for future applications. Although Ethernet has significant advantages, CAN is still the most prevalent IVN protocol, given its longer history and lower cost. In the context of this paper, we have chosen not to delve into the intricacies of Ethernet and reserved it for future work. The CAN message format is depicted in Figure 2.

Modern vehicle networks can contain more than 100 ECUs, with most ECUs implementing a UDS server. A large number of ECUs adds much weight, cost, and complexity to a vehicle. The automotive industry is now undergoing a major shift in functionality with a new concept called the Software-Defined Vehicle (SDV), heralding a transformation approach to new automotive design and functionality by transitioning from hardware-centric models to those dominated and defined by software. This notion has emerged due to advancements in automotive technologies, particularly with the intensification of electrification, connectivity, and autonomous driving capabilities within vehicles. The underlying premise resides in utilizing software as the primary medium to control, optimize, and innovate vehicular functionalities and user experiences. With a consolidation of ECUs, fewer ECUs will implement UDS with possibly fewer variance among the deployed UDS implementations.

Unified Diagnostic Services (UDS)

UDS is one of the most prominent application layer protocols in an IVN. It presents a client-server model, where a client submits queries to a server in the form of "requests" and the server replies in the form of "responses". Usually, the client is the tester and the server is an ECU, but an ECU can also act as a client and make internal UDS requests to other ECUs. UDS is standardized in ISO 14229 [10].

UDS operates as a request-driven protocol and is adaptable for implementation over various foundational protocols, including CAN and Ethernet. Figure 1 illustrates potential UDS configurations. This paper focuses on the most common version, UDS on CAN.

Designed for diagnostics purposes, UDS is a strong tool with great capabilities. The UDS standard defines many "services", protocols for different purposes, from which OEMs (Original Equipment Manufacturer) can selectively integrate into each ECU. The capabilities of these services can range from simply communicating that the connection is still ongoing using the *Tester Present* service, to reading and writing data by memory addresses using *Read Memory By Address* and *Write Memory By Address* services, and to resetting the ECU

FIGURE 1 IVN Architectures, 7 Layer OSI Model (Adapted from [7])

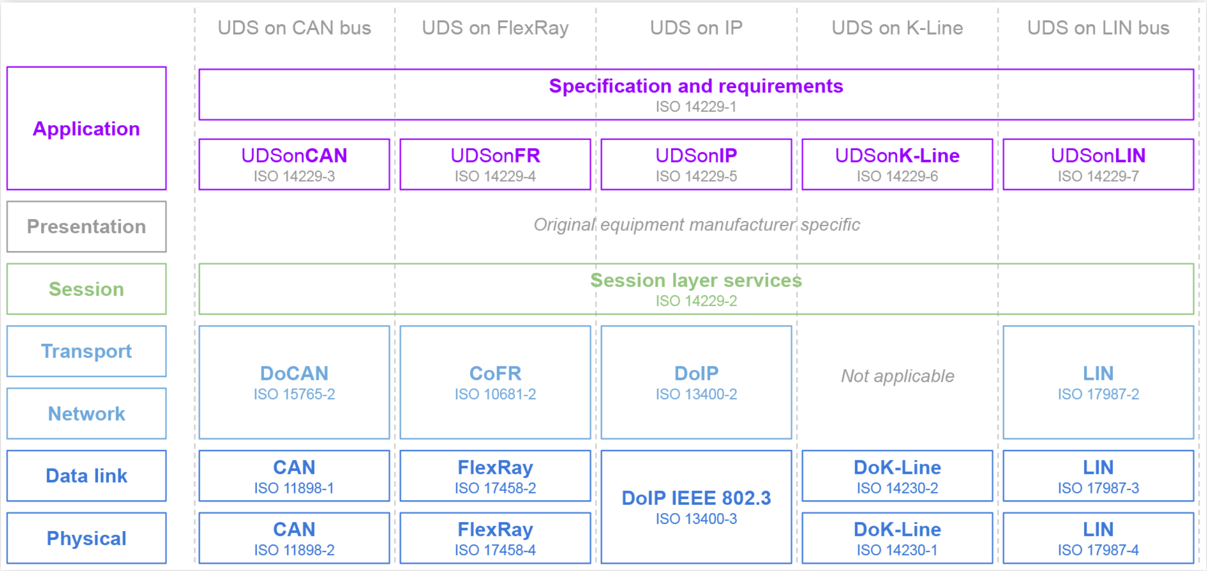
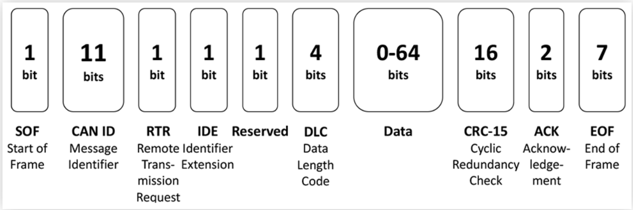


FIGURE 2 CAN Message Format (Source: [9])



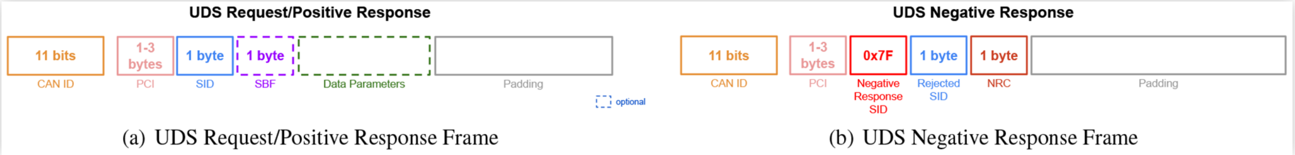
with the *ECU Reset* service. Which services each ECU defines may be different, even inside the same vehicle, and is left to the OEM's discretion. While the UDS standard outlines the basic message structures and service identifiers, OEMs typically determine the specific syntax and significance of messages, as well as the encoding methods for return values. As a result, this information is proprietary and not publicly documented [11].

The basic structure of a UDS request message consists of four notable fields: *CAN ID*, *Protocol Control Information (PCI)*, *Service Identifier (SID)*, *Sub Function Byte (SFB)*, and *Data Parameters*.

A positive response to a UDS request is very similar and has the same structure. Figure 3(a) displays the structure of a UDS request/positive response message.

- **CAN ID**: An 11-bit ID associated with every CAN message. This field has a slightly different meaning in UDS compared to raw CAN. The CAN protocol does not mandate the assignment of a unique ID to each ECU; rather, it employs CAN IDs for message prioritization and identification. UDS, on the other hand, correlates each CAN ID with a distinct ECU to establish a source/destination mechanism. Within this framework, each UDS server listens to a certain set of CAN IDs and issues responses from a singular CAN ID.
- **PCI**: PCI is a 1 to 3 bytes long field containing information about the length of the message. Up to 8 bytes of data can be sent by a single CAN packet, which is further reduced by the PCI, SID, and SBF bytes' occupying space; but, messages containing larger data can be transmitted over multiple CAN packets. This field is identified not by the UDS standard, but the ISO-TP standard [12] instead, which defines the transportation layer of UDS over CAN.
- **SID**: The request and response of each UDS service is assigned a unique ID by the ISO 14229 specification [10]. Typically, the SID of a positive response is *0x40* greater than its corresponding request ID. For instance, the *Diagnostic Session Control Service* request, designed to modify the diagnostic session mode, has the ID *0x10*; and, the positive response to this request is *0x50*.
- **SBF**: Some UDS services may define "subfunctions" to specify the exact use of the service. In other words, it allows a single service to have different functionalities or configurations. Continuing the example, an SBF of *0x2* combined with the SID *0x10*

FIGURE 3 UDS Message Structure (Adapted from [7])



(a) UDS Request/Positive Response Frame

(b) UDS Negative Response Frame

signals a request to change the diagnostic session mode to "Programming Mode". Conversely, an SBF of *0x1* indicates a request to switch to the "Default Mode".

- **Data Parameters**: Defines the payload of the package. Some services or sub functions may not require any data all; *Diagnostic Session Control* is one such example.

The structure of the negative response differs slightly. It still starts with the CAN ID and PCI field, but introduces two new fields: *Rejected SID*, and *Negative Response Code (NRC)*. It also no longer contains an SBF or data parameters. Figure 3(b) displays the structure of a UDS negative response message. A negative response is identified by its unique SID, which is defined as *0x7F*.

- **Rejected SID**: The SID of the rejected UDS request. Replaces the SBF field.
- **NRC**: Contains information about why the request was rejected. Some common ones include *0x11* for *Service Not Supported*, and *0x33* for *Security Access Denied*. Replaces data parameters.

Due to its robust features, UDS presents a significant target for potential attackers. With malicious intent, UDS can be used to steal data or bypass business operations [13], as a gateway for denial of service attacks, to disrupt software updates [3], even for sabotage or for gaining remote access to the vehicle [2, 5, 13]. Given that UDS operates on networks like CAN, which lack built-in security, it necessitates its own security mechanisms. Three UDS services are defined for this purpose:

Security Access (0x27): Provides access to security critical services. Clients cannot access some of the services that ECUs offer without first gaining security access using this service. Which services are locked behind security access is left to the discretion of the OEM. The basic structure of this protocol is as follows: First, the client requests a seed (a cryptographic nonce) from the server. The server decides on a seed, and sends it to the client. Then, both the client and the server calculate a key using the seed and a pre-determined algorithm. Lastly, the client sends the key to the server, and the server authorizes security access if the key it received matches the key it calculated. How the nonce is decided and how the key is calculated using the nonce is not defined by the UDS standard. This design is subject to numerous vulnerabilities. First, the UDS standard has no requirements on the length of the seed. Second, it has no requirements on the cryptographic strength of the algorithms used. Third, this service does not provide authenticated or encrypted communication; hence, it can be targeted by a variety of attacks [1, 2, 3] present real-world applicable attacks to UDS by exploiting this service.

Authentication (0x29): Presented in the 2020 edition of [10] as an improvement and an alternative to 0x27, 0x29 provides much better security compared to 0x27. It identifies different methods of authentication, most notably through PKI certificate exchange. Aside from authentication, it optionally provides a means for key generation through Diffie-Hellman key agreement algorithm. However, it is not yet widely implemented in real-world scenarios, and is not implemented in any of the open-source frameworks we have tested.

Secured Data Transmission (0x84): Provides confidentiality and integrity to UDS messages through means of encapsulation. This service is used encrypt and/or sign another UDS packet, which it includes in its data parameters. It allows ECUs to choose between different algorithms to be used in encryption and signature schemes, and dynamically determine signature sizes. Seemingly, there is not a lot of information regarding this protocol, aside from the specifications in ISO 14229.

Fuzz-Testing

Fuzz testing has been identified as a highly effective methodology for testing security vulnerabilities, especially in the automotive industry [14, 15] state that fuzzing and penetration testing technologies should be applied in the development of automotive cybersecurity activities [16]. The primary objective of fuzz testing is to systematically generate a substantial volume of unanticipated input using a certain methodology, and observe the system's response in order to identify potential vulnerabilities. Fuzzing techniques are mainly categorized based on input construction methods, which can be classified into two main types: Mutation-based fuzzing and generation-based fuzzing. Based on the tool's knowledge of the system under test, it is further labeled as white box, black box, or gray box [17].

Mutation-based fuzzing involves modifying existing valid inputs by randomly mutating or combining certain parts of the previous inputs [18]. It allows for the creation of smaller fuzz input sets, but its success depends on the diversity and coverage of the starting set. On the other hand, generation-based fuzzing focuses on creating entirely new inputs from scratch through specific rules or models. It allows for the creation of a more diverse set of inputs, enabling the discovery of complex vulnerabilities that may be difficult to find through mutation-based fuzzing alone; however, this leads to a larger set of fuzz inputs, leading to the tests taking a longer time. The ratio of vulnerabilities discovered to test inputs generated is expected to be larger with the former technique; in contrast, the amount of vulnerabilities discovered is expected to be higher with the latter. We use both techniques to an extent, as discussed in **PCAN-View** and **Comparison of Fuzzers**.

Black-box testing treats the system as a "black box" with no knowledge of its internal workings. It focuses on the system's inputs and outputs with no regard to the source code or the internal structure. It tests the system with real-world usage scenarios to assess how well the software meets its intended goals. Black-box testing is invaluable for evaluating the overall functionality, performance, and compliance of any software system; because it does not require any specific knowledge and the same testing software can be easily transferred to other systems. White-box testing involves examining the source code, control and data flow, as well as the logic of software components, thereby providing deep insights into the software's behavior. It may uncover problems specific to the system under test. However, it requires access to the source code and in-depth technical knowledge and may not uncover issues related to system integration or external interfaces. Also, it requires the tests to be specific to a system and the testers generated this way are not very transferable. Gray-box testing involves a combination of both methods: A high-level system knowledge and some understanding of the internal components, but not a deep understanding of the source code. Testers then use these information to design effective test scenarios that focus on specific parts of the system.

Fuzz testing can be applied in two different prominent styles with distinctive approaches and objectives:

Source code fuzzing targets the code base of the system or application under test by introducing malformed or unexpected input data directly into functions and code syntax. This technique is considered to be very similar to unit tests. One of the well-known fuzz testing tools in the cybersecurity realm is *American Fuzzy Lop* (AFL) [19], which is among the most widely used fuzzers [20].

Protocol fuzz testing predominantly targets the exploration of vulnerabilities in the communication protocols used by software applications. This technique uses a certain protocol, such as UDS, to communicate with the system to test for system robustness. One widely recognized tool specifically tailored for protocol fuzz testing is a commercial tool named Defensics [21], which we also employ in our tests.

Both protocol fuzz testing and source code fuzzing hold imperative positions in a comprehensive security testing strategy. Source code fuzzing should be conducted early on in the software development life cycle to catch issues as they arise. Protocol testing should periodically be employed in major software releases to test for potential vulnerabilities.

Related Work

UDS

Engaging with UDS is notably challenging, not only because of its complexity, but also because relevant resources are scarce, much of the information is proprietary, and obtaining the standard is expensive [7] offers a strong introduction to the subject.

As in-vehicle electronics continue to advance and new vulnerabilities are discovered, the UDS protocol is continually evolving. Due to the relative novelty of *0x29: Authentication Service*, which was added to the last iteration of the UDS standard [10] in 2020, academic research has predominantly concentrated on the *0x27: Security Access Service*.

Van den Herrewegen *et al.* [2] studied vehicles from four major automotive manufacturers. It targets 0x27 and exposes several vulnerabilities related to both implementation processes and cipher selection. After gaining security access through attacks, it demonstrates how an attacker can recover secrets or run malicious code in the ECU.

Sermpinis [1] discovers a vulnerability in the random seed generation of ECUs, which is used for *Security Access Service*, and employs fuzz testing techniques to verify and attack the vulnerability.

Lauser *et al.* [4] use the Tamarin Model [22] to formally analyze the security of both 0x27 and 0x29. It argues that the prior lacks details in the standardization, which leads to insecure implementations; and, discovers two vulnerabilities in the latter. It concludes by noting that even though the standard might be secured, the implementations must also be checked for errors.

UDS Fuzzing

UDS Fuzzing has received limited attention in literature. Sermpinis [1] employs fuzzing techniques to highlight a critical vulnerability of a proprietary UDS implementation used in real-world situations. The presented attack abuses the weak random number generation in ECUs, and the fact that the *ECU Reset* service is not locked behind security access. The works in [13, 23, 24] advocate for fuzz testing as a potent instrument for automotive security, with an emphasis on the CAN bus. Notably, [13] integrates the suggested fuzzer through UDS services. Patki *et al.* [25] design another CAN fuzzer and compare it to existing proprietary automotive fuzzers. Luo *et al.* [16] delve into methods for automotive testing and propose a penetration testing software providing both a CAN fuzzer and a specialized UDS fuzzer.

Two prevalent themes are observed from previous work. First, they focus on fuzzing ECUs while using CAN or UDS as a gateway; and second, they fuzz proprietary ECU implementations. We differ from them by directly targeting open-source UDS implementations.

Experimental Setup

Our experimental setup consists of a server running on a Linux system and a client running on Windows 10. A simple two-device CAN network is established between the server and the client using two PCANUSB connectors [26] connected with a breadboard. The tested opensource UDS servers are deployed one by one on the Linux environment. The fuzzers (Caring Caribou, Defensics), as well as PCAN-View are used from another computer running Windows 10. PCAN-View is mainly used to manually send CAN messages and to sniff the CAN traffic generated by the testing software and the server under test. We use the following tools in our test setup:

Caring Caribou

Caring Caribou [27] is an open source security testing software designed to be deployed as a black-box tool on any CAN network. It was used by [1] to find a vulnerability in a real ECU and to attack it. It is a modular and expandable tool that can be used as a basis for commercial fuzzers. Although Caring Caribou provides many functionalities, we particularly use the *fuzzer* and *uds* tools. The *fuzzer* tool allows fuzzing the UDS servers through different fuzzing methods; and, the *uds* tool provides several helpful functionalities. We use the former with the mutation option to set a specific CAN ID and/or an SID while mutating other bits, and the latter with *discovery*, *services*, and *subservices* options. These options discover the CAN IDs to which servers listen to and from which they respond, detecting the services that they provide, and the subservices that each service encapsulates, respectively.

The Bash snippet depicted in Listing 1 showcases the output of Caring Caribou run with the *uds services* option, discovering the services that an UDS server named Gallia [28] exposes.

LISTING 1: Output of Caring Caribou

```
$ cc.py uds services 0x001 0x009
------------------
CARING CARIBOU V0.4
------------------
Loading module 'uds'
Probing service 0xff (255/255): found 37
Done!
Supported service 0x01: Unknown service
Supported service 0x02: Unknown service
Supported service 0x03: Unknown service
Supported service 0x04: Unknown service
Supported service 0x05: Unknown service
Supported service 0x06: Unknown service
Supported service 0x07: Unknown service
Supported service 0x08: Unknown service
Supported service 0x09: Unknown service
Supported service 0x0a: Unknown service
Supported service 0x10:
    ↪ DIAGNOSTIC_SESSION_CONTROL
Supported service 0x11: ECU RESET
Supported service 0x14:
    ↪ CLEAR_DIAGNOSTIC_INFORMATION
Supported service 0x19: READ DC INFORMATION
Supported service 0x22: READ DATA BY
    ↪ IDENTIFIER
Supported service 0x23: READ MEMORY BY ADDRESS
Supported service 0x24:
    ↪ READ_SCALING_DATA_BY_IDENTIFIER
Supported service 0x27: SECURITY ACCESS
Supported service 0x28: COMMUNICATION CONTROL
Supported service 0x29: Unknown service
Supported service 0x2a: READ DATA BY PERIODIC
    ↪ IDENTIFIER
Supported service 0x2c: DYNAMICALLY DEFINE
    ↪ DATA IDENTIFIER
Supported service 0x2e: WRITE DATA BY
    ↪ IDENTIFIER
Supported service 0x2f:
    ↪ INPUT_OUTPUT_CONTROL_BY_IDENTIFIER
Supported service 0x31: ROUTINE CONTROL
Supported service 0x34: REQUEST DOWNLOAD
Supported service 0x35: REQUEST UPLOAD
Supported service 0x36: TRANSFER DATA
Supported service 0x37: REQUEST TRANSFER EXIT
Supported service 0x38: REQUEST FILE TRANSFER
Supported service 0x3d:
    ↪ WRITE_MEMORY_BY_ADDRESS
Supported service 0x3e: TESTER PRESENT
Supported service 0x83:
    ↪ ACCESS_TIMING_PARAMETER
Supported service 0x84: SECURED DATA
    ↪ TRANSMISSION
Supported service 0x85: CONTROL DIC SETTING
Supported service 0x86: RESPONSE_ON_EVENT
Supported service 0x87: LINK CONTROL
```

Defensics

Defensics [21] is a multi-purpose, proprietary black-box fuzz tester. Although we used it to test UDS and CAN, it contains modules to test various protocols from other industries. It uses a method called *instrumentation*, which is querying a simple pre-set input before and after the test input. By sandwiching the test input in between two other inputs, the program can detect the exact input that causes the server to misbehave. Aside from being a fuzzing tool, Defensics also offers other testing methods as well, but they are out of our scope. Convenient for non-experts, it has a relatively simple graphical interface. Figure 5 depicts an example run of Defensics.

SocketCAN

SocketCAN or can-utils [29] is a set of tools and utilities to access a CAN network. Among the several utilities that it provides, we particularly make use of *candump* and *cansend*, which are used to log and display the CAN traffic, and send custom CAN messages, respectively.

PCAN-USB CAN Bus Connector

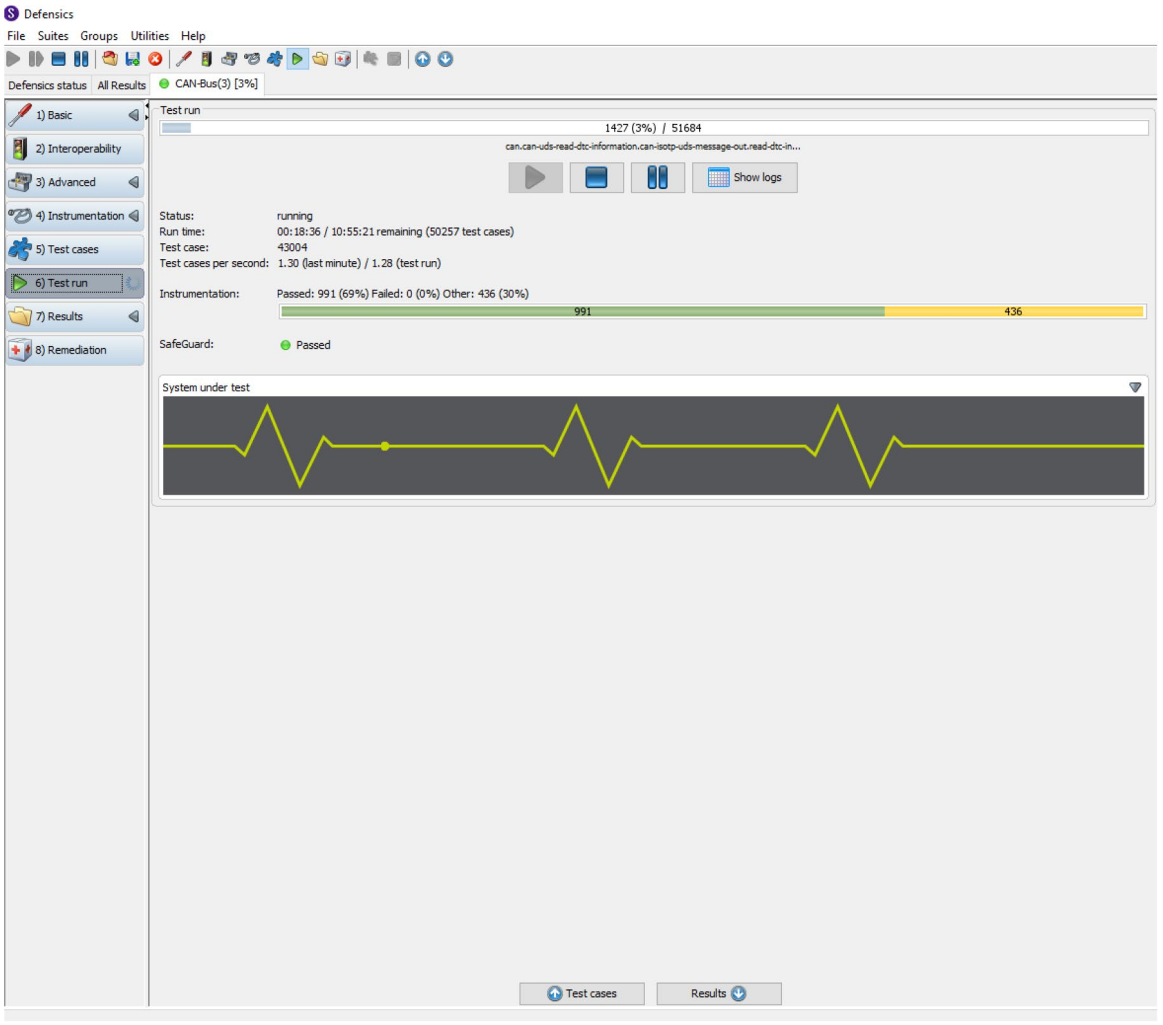
The PCAN-USB [26] is a straightforward tool that offers a USB interface for connecting a computer to a CAN network. The adapter provides CAN drivers as well, which Windows devices typically lack. We employ two of them to connect two computers to a CAN network. The appearance of the tool is illustrated in Figure 4.

PCAN-View

PCAN-View [30] is a Windows GUI tool with capabilities to sniff, send, and record messages in a CAN network. Being a graphical tool makes it more user friendly

FIGURE 4 PCAN-USB CAN Bus Connector

FIGURE 5 Example Run of Defensics



1427 (3%) / 51684

can.can-uds-read-dtc-information.can-isotp-uds-message-out.read-dtc-in...

Status: running
Run time: 00:18:36 / 10:55:21 remaining (50257 test cases)
Test case: 43004
Test cases per second: 1.30 (last minute) / 1.28 (test run)

Instrumentation: Passed: 991 (69%) Failed: 0 (0%) Other: 436 (30%)
991 436

SafeGuard: ● Passed

compared to terminal applications, especially for non-experts. The software is included with PCAN-USB.

Methodology

Overview

We test each UDS implementation in a virtual setting; that is, we do not deploy them on an actual ECU for testing. This is because writing to random memory addresses or setting invalid but unchecked parameters can permanently damage an ECU or even render it inoperative. Consequently, conducting tests in a virtual environment is not only more cost-efficient but also prevents irreversible harm to the hardware.

We sequentially deploy the UDS implementations on a Linux system. First, following a black-box approach, we test the capabilities of the implementations by sending simple UDS packets and listening to responses. We initiate this process by employing the *uds* tool of Caring Caribou to discover the services, subservices, and CAN IDs that each library exposes. Following this, we transmit UDS request packets — deliberately structured to elicit specific positive or negative responses from the server—and analyze the received responses or their absence to draw conclusions. Second, we use the fuzz testing tools, namely the *fuzzer* module of Caring Caribou and Defensics. We compare the two tools as well as the UDS servers under test.

Deploying the fuzzer directly on the implementations to systematically uncover vulnerabilities presents a non-trivial challenge when the systems under test are mostly unfinished work. This is primarily because of two reasons. First, among the implementations surveyed, only two include the *ECU Reset Service*, which is essential for continuous fuzzing. Notably, among the two libraries that implement the *ECU Reset Service*, one ceases to respond following an *ECU Reset* request. Second, aside from being unfinished work, several of the implementations are designed as libraries, and the example programs they implement hold placeholder functionality, if any. Thus, uncovering any “hidden” vulnerabilities is made harder by many surface level issues.

Metrics

We evaluate the UDS implementations in two aspects: The number of services and subservices they provide, and the correctness of their behavior. We use Caring Caribou to uncover the services and subservices; and, use Caring Caribou together with Defensics to fuzz the UDS servers to test their behavior. We also compare the two tools.

Number of Services

We employ Caring Caribou's *uds* module to determine the number of services each library implements; and review the documentation and the code to identify any partially implemented service that the tool might have missed. It should be noted that such an oversight can only occur in instances where the server fails to respond to a request, or the example program does not include all implemented services. We then manually check each service through UDS communication to better understand to which extent it was implemented, e.g., whether changing the diagnostic session with *Diagnostic Session Control* has any real effect to the programs' behavior, how the seed is generated for *Security Access*, etc.

Defensics could also be employed to discover the available services, but Caring Caribou conducts a more extensive brute search while Defensics searches for known services. Caring Caribou is also faster due to not employing instrumentation, which is not necessary for the discovery task.

Correct Behavior

First, we employ Caring Caribou's *fuzzer* module to generate random inputs for the server. We evaluate the servers' robustness by submitting both coherent and incoherent messages, observing whether such inputs induce system instability, unresponsiveness, or failure. The test cases generated this way are valid CAN frames, but they are not generated according to UDS specifications. This module does not listen to server responses either, which adds another layer of difficulty to testing for incorrect behavior. Caring Caribou also offers a *uds fuzz* module which does listen to server responses; however, that module serves the purpose of testing the *Security Access Service* as employed by [1], and not the general behavior of the system under test.

Secondly, we employ Defensics, which can generate fuzz test inputs specifically to test the UDS server behavior. Instead of rapidly generating test cases regardless of server behavior, Defensics checks the server response before and after every test input; and, it generates test cases specifically to cover the UDS layer [31].

Comparison of Fuzzers

Caring Caribou is built on top of *python-can* and allows the user the freedom to extend it as needed since it is fully written in Python and is open sourced. Aside from some specific applications for the *Security Access Service*, Caring Caribou lacks any knowledge of how to send correct UDS fuzz messages based on ISO 14229; so many invalid inputs will be dropped by the target before reaching deep in the target's code paths, reducing the impact of fuzzing and increasing test times. It can be deployed trivially onto any CAN network, and is still actively developed and maintained.

Defensics is a commercial tool and has contributed to many CVEs over the years. It is capable of generating fuzzed messages based on ISO 14229 [10], which increases the chance of finding a failure, since it can test more code paths in the system under test by specifically targeting different behavior from the target. It is also useful for non-UDS related testing since it contains a large selection of test suites with varying purposes. It must also be noted that Defensics requires a PCAN interface and does not work on virtual CAN networks out of the box.

In summary, Defensics is a robust tool with applicability across various domains, while Caring Caribou offers a lightweight, extensible and modular command-line interface that can interact with any CAN interface through SocketCAN, including virtual CAN interfaces.

Experimental Evaluation

We identified eight open-source UDS server implementations available on GitHub, among which two were designed for special hardware: PASTA [32] and RAMN [33] are vehicle testbed projects that include UDS server implementations. We focus on the six that is not platform specific, but still mention the other two; because, they are open sourced and may provide valuable starting points for more general UDS implementations, especially given that they are already used in real-world scenarios together with their hardware counterparts.

A server may *expose* and/or *implement* a UDS service, if the server recognizes the service and/or contain functionality for it, respectively. Among the implementations we found, four implement and expose some subsets of services, as can be seen in Table 1. More details about the exact services each library implements or exposes are depicted in Table 2 (see APPENDIX A).

Gallia

Gallia [28] is an extendable automotive penetration testing framework. It can function as a UDS server—or in Gallia's terminology—a "virtual ECU". Among the implementations we've identified, Gallia stands out as the most robust and dependable by a significant margin. Moreover, it

TABLE 1 Number of UDS services implemented by projects.

Project Name	Services Exposed	Services Implemented
Gallia [28]	27[1]	10
iso14229 [34]	13	13
uds-to-go [35]	5	5
UDSim [36]	24[1]	0
py-uds [37]	0	0
UDS-Server [38]	0	0

[1] As can also be seen from Listing 1 Gallia exposes "services" with SID *0x01* through *0x0a*. These are OBD-II functionality and are not UDS services. Similar holds for UDSim.

comes with built in support for CAN and TCP as the underlying layer of communication.

Gallia provides the option to generate a virtual ECU with randomized services and subservices, as well as the option to simulate an ECU from a database of UDS logs. Our focus was on the first option, through which one can modify the randomness and generate a functioning UDS server which exposes any common UDS service. In fact, it is the only stack that provides actual functionality for the services it implements; thus, it was the only stack upon which we found it meaningful to deploy Defensics.

It proved reliable against Caring Caribou's CAN fuzzer module, as well as Defensics' UDS fuzzer suite. Indeed, it passed all the tests generated by Defensics.

We must also note that as of writing this paper, Gallia has a very active GitHub community, which could prove useful for new contributors.

iso14229

iso14229 [34] implements both client and server UDS applications for Linux environment. Its main purpose is to serve as a library for UDS function calls instead of a standalone UDS implementation; however, it provides a simple server and client example as well. These examples mainly hold placeholder functionality for the services. For instance, the *Security Access Service* returns an example seed of [1, 2, 3, 4], and grants access to any key. Regardless, it implements 13 services in total and is one of the easier-to-use implementations.

Although it had proven to be mostly reliable, sending a simple *ECU Reset* request resulted in the example server to go into an unresponsive state. Additionally, since it primarily featured placeholder functionality for the services implemented, using Defensics on it did not yield any significant results.

uds-to-go

uds-to-go [35] is a lightweight UDS server and client implementation. We have discovered that as of writing this paper, it implements five services; however, sending a *Read Data By Identifier* request caused the server to stop responding.

UDSim

UDSim [36] is an ECU network simulator. Most notably, it comes with a GUI to simulate the network traffic visually. Its main purpose is to listen to UDS data through network sniffing or by reading UDS logs, and then simulating the network. It recognizes all standard UDS messages, but does not respond to them. After training on real UDS communication, it can function as a UDS server. However, it is not immediately deployable as a standalone UDS server without training, unless it is extended with the functionality.

py-uds

py-uds [37] is still in its initial stages of development. While it does not currently have a UDS server implementation, it is intended to implement support for both the client and server aspects of UDS across CAN, Ethernet, LIN, FlexRay, and K-Line.

UDS-Server

Finally, UDS-server [38] is a Scala implementation for UDS. The repository lacks documentation and instructions for building the application, preventing us from testing it. However, we chose to mention it in our paper since it represents the sole Scala-based UDS implementation we came across. Reading the code manually, we believe that it does not implement any services.

Conclusion

History has repeatedly demonstrated that effective software security results originate from collaborative efforts, and hindering such collaboration through obscurity is detrimental to security systems. Vehicle security is no exception. For UDS to garner interest from a broader spectrum of researchers, it needs to be more publicly accessible. One potential avenue to facilitate this is the establishment of robust open-source libraries. Despite our extensive search, we could only identify six non-platform specific open-source UDS server implementations. Our examination and fuzzing of these implementations revealed that many are preliminary projects stemming from the efforts of individual researchers; moreover, there is not a single project that implements more than half of the services. Nonetheless, certain notable implementations present significant potential to make the field of UDS security more accessible to emerging researchers, and to be a good foundation for real-world applications. Notably, the Gallia framework [28] distinguished itself with its thorough and robust implementation of a virtual ECU, while iso14229 [34] emerges as a commendable runner-up.

Robust UDS implementations require proper testing, especially for the purpose of security-related services and functionalities. The capabilities of current open-source automotive penetration testing frameworks have yet to match the advanced features offered by leading commercial tools. We argue that more advanced open-source fuzzers are a necessity for extensive testing in the automotive domain for reasons similar to the necessity of open-source UDS implementations.

Moving forward, we plan to contribute to the landscape of open-source UDS to bring it to a state comparable to real-world ECUs, by improving upon the existing UDS repositories and fuzzing tools by actively participating in their development.

References

1. Sermpinis, T., *Uds Fuzzing and the Path to Game Over* (Heidelberg, Germany: Presented at the Troopers, 2022)
2. Van den Herrewegen, J., and Garcia, F., "Beneath the Bonnet: A Breakdown of Diagnostic Security," in *23rd European Symposium on Research in Computer Security, Esorics 2018*, Barcelona, Spain, September 3-7, 2018, Lecture Notes in Computer Science. Springer, Cham, Aug. 2018, vol. 11098, 305-324, doi: 10.1007/978-3-319-99073-6_15.
3. Liis, J., "Security Evaluation of the Electronic Control Unit Software Update Process," Ph.D. dissertation, Royal Institute of Technology, 2014.
4. Lauser, T., and Kraus, C., "Formal Security Analysis of Vehicle Diagnostic Protocols," in *Proceedings of the 18th International Conference on Availability, Reliability and Security* (Benevento Italy: ACM, 2023), 1-11, 10.1145/3600160.3600184 (visited on 10/09/2023).
5. Greenberg, A., "Hackers Remotely Kill a Jeep on the Highway—With Me in It," Wired, 2015, accessed October 9, 2023, https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.
6. Khatri, N., Shrestha, R., and Nam, S.Y., "Security Issues with In-Vehicle Networks, and Enhanced Countermeasures Based on Blockchain," *Electronics*, 10, 8, 893, 2021, 10.3390/electronics10080893 (visited on 10/09/2023).
7. Martin Falch, "UDS Explained - A Simple Intro," accessed: October 9, 2023, CSS Electronics, 2022, https://www.csselectronics.com/pages/uds-protocol-tutorialunified-diagnostic-services.
8. "Road Vehicles — Controller Area Network (CAN) — Part 1: Data Link Layer and Physical Signalling," International Organization for Standardization, Geneva, CH, Standard, 2015.
9. Pesé, M.D., "Bringing Practical Security to Vehicles," Ph.D. dissertation, The University of Michigan, 2022.
10. "Road Vehicles—Unified Diagnostic Services (UDS)—Part 1: Application Layer," International Organization for Standardization, Geneva, CH, Standard, February 2020.
11. Yu, L., Liu, Y., Jing, P., et al., "Towards Automatically Reverse Engineering Vehicle Diagnostic Protocols," in *Proceedings of the 31st USENIX Security Symposium*, USENIX Association, 2022, 1939-1956.
12. "Road Vehicles - Diagnostic Communication over Controller Area Network (DOCAN) - Part 2: Transport Protocol and Network Layer Services," International Organization for Standardization, Geneva, CH, Standard, April 2016.
13. Bayer, S., and Ptok, A., "Don't Fuss about Fuzzing: Fuzzing Controllers in Vehicular Networks," 2015, https://www.escar.info/images/Datastore/2015_escar_EU_Papers/3_escar_2015_Stephanie_Bayer.pdf.
14. SAE International, "Cybersecurity Guidebook for Cyberphysical Vehicle Systems," SAE Standard J3061 202112, December 2021, STABILIZED Dec 2021.
15. "Road Vehicles - Cybersecurity Engineering," Geneva, Switzerland, Tech. Rep. 21434, August 2021, International Standard published [60.60], 81.
16. Luo, F., Zhang, X., and Hou, S., "Research on Cybersecurity Testing for In-Vehicle Network," in *2021 International Conference on Intelligent Technology and Embedded Systems (ICITES)*, IEEE, 2021, 144-150. doi: 10.1109/ICITES53477.2021.9637070.
17. Dürrwang, J., Braun, J., Rumez, M., Kriesten, R. et al., "Enhancement of Automotive Penetration Testing with Threat Analyses Results," *SAE Int. J. Transp. Cyber. & Privacy* 1, no. 2 (2018): 91-112, doi:https://doi.org/10.4271/11-01-02-0005.
18. Zhang, H., Huang, K., Wang, J., and Liu, Z., "Can-ft: A Fuzz Testing Method for Automotive Controller Area Network Bus," in *2021 International Conference on Computer Information Science and Artificial Intelligence (CISAI)*, IEEE, 2021, 225-231.
19. Zalewski, M., "American Fuzzy Loop," accessed 2023-10-11, 2013, https://github.com/google/AFL.
20. Fioraldi, A., Mantovani, A., Maier, D., and Balzarotti, D., "Dissecting american fuzzy lop: A fuzzbench evaluation," *ACM Trans. Softw. Eng. Methodol.* 32, no. 2 (2023), doi:10.1145/3580596.
21. Synopsys Inc., "Defensics," accessed 2023-10-26, 2023, https://www.synopsys.com/oftware-integrity/security-testing/fuzz-testing.html.
22. Basin, D., Cremers, C., Dreier, J., et al., "Tamarin Prover," accessed 2023-10-11, 2023, https://tamarin-prover.github.io.
23. Fowler, D.S., Bryans, J., Cheah, M., Wooderson, P. et al., "A Method for Constructing Automotive Cybersecurity Tests, a CAN Fuzz Testing Example," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)* (Sofia, Bulgaria: IEEE, 2019), 1-8, 10.1109/QRS-C.2019.00015 (visited on 10/09/2023).
24. Fowler, D.S., Bryans, J., Shaikh, S.A., and Wooderson, P., "Fuzz Testing for Automotive Cyber-Security," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2018, 239-246, 10.1109/DSN-W.2018.00070.
25. Patki, P., Gotkhindikar, A., and Mane, S., "Intelligent Fuzz Testing Framework for Finding Hidden Vulnerabilities in Automotive Environment," in *2018 Fourth International*

Conference on Computing Communication Control and Automation (ICCUBEA) (Pune, India: IEEE, 2018), 1-4, 10.1109/ICCUBEA.2018.8697438 (visited on 10/05/2023).

26. PEAK-System, “PCAN-USB: Can Interface for USB, https://www.peak-system.com/PCAN-USB.199.0.html?&L=1#, Accessed: 2023-10-27, 2023.
27. GitHub Contributors, “Caringcaribou: A Friendly Car Security Exploration Tool,” accessed 2023-10-11, 2023, https://github.com/CaringCaribou/caringcaribou.
28. Fraunhofer AISEC, “Gallia - Extendable Pentesting Framework,” accessed 2023-10-26, 2021, https://github.com/Fraunhofer-AISEC/gallia.
29. Github Contributors, “Socketcan Userspace Utilities and Tools,” accessed 2023-10-11, 2023, https://github.com/linux-can/can-utils.
30. PEAK-System, “PCAN-View,” accessed 2023-10-11, 2023, https://www.peak-system.com/PCAN-View.242.0.html?&L=1.
31. Knudsen, J., and Varpiola, M., “Fuzz Testing Maturity Model,” 2017.
32. Toyota InfoTechnology Center, “Pasta: Portable Automotive Security Testbed with Adaptability,” accessed 2024-01-10, https://github.com/pasta-auto/PASTA1.0.
33. Toyota InfoTechnology Center, “RAMN (Resistant Automotive Miniature Network),” accessed 2024-01-10, https://github.com/ToyotaInfoTech/RAMN.
34. Kirkby, N.J., “ISO 14229,” accessed 2023-10-26, 2022, https://github.com/driftregion/iso14229/tree/main.
35. Andrey, A., “UDS-to-Go,” accessed 2023-10-26, 2022, https://github.com/astand/udsto-go.
36. Smith, C., “UDSIM,” accessed 2023-10-26, 2015, https://github.com/zombieCraig/UDSim.
37. Dabrowski, M., “PY-UDS,” https://github.com/mdabrowski1990/uds, accessed 2023-10-26, 2023.
38. Psyriccio, “UDS-Server,” accessed 2023-10-26, 2015, https://github.com/psyriccio/UDS-server.

Contact Information

Levent Çelik
Clemson University
lcelik@clemson.edu

John McShane
Eastern Michigan University
jmcshane@emich.edu

Iwinosa Aideyan
Clemson University
iaideya@clemson.edu

Richard Brooks
Clemson University
rrb@clemson.edu

Mert D. Pesé
Clemson University
mpese@clemson.edu

Acknowledgments

This work was supported by Clemson University’s Virtual Prototyping of Autonomy Enabled Ground Systems (VIPR-GS), under Cooperative Agreement W56HZV-21-2-0001 with the US Army DEVCOM Ground Vehicle Systems Center (GVSC).

Definitions, Acronyms, Abbreviations

CAN - Controller Area Network

UDS - Unified Diagnostic Services

LIN - Local Interconnected Network

IVN - In-Vehicle Network

ECU - Electronic Control Unit

ISO - International Organization for Standardization

ISO-TP - ISO Transport Layer

OEM - Original Equipment Manufacturer

OBD - On-Board Diagnostics

SID - Service Identifier

SBF - Sub Function Byte

SDV - Software Defined Vehicle

CVE - Common Vulnerabilities and Exposures

Appendix

TABLE 2 Services Included In Each Server Implementation

	Gallia	ISO 14229	uds-to-go	UDSim	py-uds	UDS-Server
0x10: Diagnostic Session Control	🟩	🟩	🟩	🟨	⬜	⬜
0x11: Ecu Reset	🟩	🟥	⬜	🟨	⬜	⬜
0x14: Clear Diagnostic Information	🟩	🟨	⬜	🟨	⬜	⬜
0x19: Read DTC Information	🟩	🟨	⬜	🟨	⬜	⬜
0x22: Read Data By Identifier	🟩	🟩	🟥	🟦	⬜	⬜
0x23: Read Memory By Address	🟩	🟩	⬜	🟨	⬜	⬜
0x24: Read Scaling Data By Identifier	🟦	🟨	⬜	🟨	⬜	⬜
0x27: Security Access	🟩	🟩	🟩	🟦	⬜	⬜
0x28: Communication Control	🟦	🟩	⬜	🟨	⬜	⬜
0x29: Authentication	🟦	⬜	⬜	⬜	⬜	⬜
0x2A: Read Data By Periodic Identifier	🟦	🟨	⬜	🟨	⬜	⬜
0x2C: Dynamically Define Data Identifier	🟦	🟨	⬜	🟨	⬜	⬜
0x2E: Write Data By Identifier	🟦	🟩	⬜	🟨	⬜	⬜
0x2F: Input Output Control By Identifier	🟩	🟨	⬜	🟨	⬜	⬜
0x31: Routine Control	🟩	🟩	🟩	🟨	⬜	⬜
0x34: Request Download	🟦	🟩	⬜	🟨	⬜	⬜
0x35: Request Upload	🟦	🟩	⬜	🟨	⬜	⬜
0x36: Transfer Data	🟦	🟩	⬜	🟨	⬜	⬜
0x37: Request Transfer Exit	🟦	🟩	⬜	🟨	⬜	⬜
0x38: Request File Transfer	🟦	🟨	⬜	🟨	⬜	⬜
0x3D: Write Memory By Address	🟦	🟨	⬜	🟨	⬜	⬜
0x3E: Tester Present	🟩	🟩	🟩	🟨	⬜	⬜
0x83: Access Timing Parameter	🟦	🟨	⬜	🟨	⬜	⬜
0x84: Secured Data Transmission	🟦	🟨	⬜	🟨	⬜	⬜
0x85: Control DTC Setting	🟦	🟩	⬜	🟨	⬜	⬜
0x86: Response On Event	🟦	🟨	⬜	🟨	⬜	⬜
0x87: Link Control	🟦	⬜	⬜	🟨	⬜	⬜

🟩	Implements functionality for the service
🟦	Either rejects or gives a generic response to the service
🟨	Has service data but does not respond
⬜	Does not recognize service
🟥	Implemented service fails fuzz tests

Positions and opinions advanced in this work are those of the author(s) and not necessarily those of SAE International. Responsibility for the content of the work lies solely with the author(s).

ISSN 0148-7191