2025-01-8091 Published 01 Apr 2025

LLM-Powered Fuzz Testing of Automotive Diagnostic Protocols

John McShane Eastern Michigan University

Levent Celik, Iwinosa Aideyan, Richard Brooks, and Mert D. Pesé Clemson University

Citation: McShane, J., Celik, L., Aideyan, I., Brooks, R. et al., "LLM-Powered Fuzz Testing of Automotive Diagnostic Protocols," SAE Technical Paper 2025-01-8091, 2025, doi:10.4271/2025-01-8091.

Received: 12 Sep 2024 Revised: 20 Jan 2025 Accepted: 21 Jan 2025

Abstract

Modern vehicles contain tens of different Electronic Control Units (ECUs) from several vendors. These small computers are connected through several networking busses and protocols, potentially through gateways and converters. In addition, vehicle-to-vehicle and internet connectivity are now considered requirements, adding additional complexity to an already complex electronic system.

Due to this complexity and the safety-critical nature of vehicles, automotive cyber-security is a difficult undertaking. One critical aspect of cyber-security is the robust software testing for potential bugs and vulnerabilities. Fuzz testing is an automated software testing method injecting large input sets into a system. It is an invaluable technique across many industries and has become increasingly popular since its conception. Its success relies highly on the "quality" of inputs injected.

One shortcoming associated with fuzz testing is the expertise required in developing "smart" fuzz testing tools (fuzzers). Developing a fuzzer requires expertise on various topics, from input types and underlying networks to potential system configurations. Moreover, fuzzers are generally not transferable between different systems, limiting their reuse. This study investigates whether Generative AI technologies can meaningfully assist in their development by comparing an AI-generated fuzzer against a commercial one.

An automotive fuzzer focusing on Unified Diagnostic Services (UDS) was developed by exclusively querying an AI model. First, the pre-trained AI is taught the underlying structure and constraints of UDS and is then used to generate semantically valid test cases. The effectiveness of test cases for vulnerability and fault detection is evaluated. The impact of specific queries and the underlying protocol network configurations on the generated test cases is then investigated through comparison with a commercial fuzzer.

Introduction

Vehicles are rapidly becoming more connected, and their electronics are increasingly complex. While this transformation dramatically enhances the user experience, it also expands the vehicles' attack surface. Automotive software security is challenging due to the inherent time-sensitivity of in-vehicle communication and the low computational power of electronic control units (ECUs) – individual computing units of a vehicle. Recent automotive security regulations have established mandatory cybersecurity standards, which makes it vital for automotive manufacturers and suppliers to develop secure vehicle software [1].

UDS (ISO 14229) [2] is a standardized diagnostic protocol. Its functionality goes beyond simple diagnostics; it provides authentication and data security mechanisms and even allows updating ECU firmware; as a result, it is a prime target for cyber attacks.

Fuzz testing is an automated testing methodology that generates a sequence of test inputs, either randomly or algorithmically, to identify unexpected program behavior. It is often used in penetration testing to uncover bugs and vulnerabilities but does not replace a full penetration test's broader scope and objectives. The size of the input set is often exponential on input length. Thus, the approach to generating the inputs is the main characteristic of a fuzzer because it determines the test's length and effectiveness.

We employ a generative AI (GenAI) to model a UDS fuzzer to achieve a higher effectiveness-to-test-time ratio. Two broad research questions can be formalized when looking into GenAI and a fuzz testing framework:

1. How can AI be deployed to enhance the feedback during a fuzz testing run and adjust to a target dynamically?
2. Can AI be used to model protocol-based fuzz testing to require a lower degree of manual input?

Our work focuses on the second question and develops a fuzzer tool with the help of AI. We do use AI

to generate test cases statically, but we do not utilize a feedback loop. Using AI in a feedback loop for runtime adjustments is beyond the scope of this paper and is left for future work. Specifically, we identify three intellectual contributions (ICs):

IC-1: Development of a UDS Fuzzer Model

We reveal that a large language model (LLM) can generate the code for a fuzzer by utilizing structured prompts, including parts from ISO/SAE standards or log traces. However, we note that UDS is a relatively simple protocol, and whether the same success can be achieved for more complex protocols such as TLS or Bluetooth has not been investigated.

IC-2: Effectiveness in Vulnerability Detection

We demonstrate the effectiveness of fuzz testing, evaluated in the number of vulnerabilities uncovered and the total test time, by finding a critical vulnerability in a device under test (DUT) in production. We compare the traditional fuzz testing methodologies against our fuzzer generated by an LLM by deploying them onto the same program.

IC-3: LLM Created Test Cases

Generating test cases in an efficient and targeted manner to quickly uncover vulnerabilities is a key factor in determining the effectiveness of a fuzzer. Given their capabilities, AI algorithms naturally present a strong option for improving fuzzing techniques. This work examines the impact of using an LLM to generate test inputs.

Background

In-Vehicle Network Basics

Many low-layer protocols are standardized for in-vehicle networks (IVNs), among which Controller Area Network (CAN) bus is widely used for safety-critical tasks with real-time requirements. CAN is a message-based broadcast protocol with an internal priority mechanism. Most internal messages are sent over CAN, such as the communication between the engine and the accelerator pedal. While CAN is the primary focus of this paper, other networks like Ethernet are also widely used in automotive applications.

A CAN frame is depicted in Fig. 1. It consists of several fields, notably, an 11-bit identifier, up to 8 bytes of data, and a Cyclic Redundancy Check (CRC) for accidental errors. The identifier serves as a "message identifier", declaring the meaning as well as the priority of messages [2].

Although 8 bytes of payload is sufficient for many in-vehicle communication purposes, it becomes a significant limitation for diagnostics, which includes firmware updates of much larger file sizes. This limitation is overcome with the ISO-TP protocol (ISO 15765-2) [4], which defines the transport layer. A message can be divided into several CAN frames through ISO-TP and sequentially transmitted.

UDS

UDS, depicted in Fig. 2, defines the application and session layers of an IVN. It standardizes diagnostic messages' basic syntax and significance but leaves many implementation details to the original equipment manufacturer (OEM). Its agnosticism of the underlying networks allows it to be implemented on several different lower-layer architectures and enables us to focus only on CAN. UDS operates on a request/response basis within a client/server model, and it has several use cases named "services". UDS requests and responses, depicted in Fig. 3, are very similar in syntax with three notable fields:

- **CAN ID:** The identifier of CAN serves a different purpose in UDS. Instead of serving as a message identifier, it provides a source mechanism by assigning a unique ID for ECUs and a range of tester IDs.
- **SID (Service ID):** A single-byte service identifier.
- **SBF (Sub-Function Byte):** Some UDS services have an optional sub-function byte to customize the service; e.g., *Security Service* uses the *SBF* field to define the level of security access.

The structure of negative responses is slightly different. As illustrated in Fig. 4, two new fields are introduced, and the *SBF* and *Data Parameters* fields are discarded.

- **Negative Response SID**: Indicates that the response is negative and is defined by the standard as always being *0x7F*.

FIGURE 1 CAN frame breakdown (Source: [3])

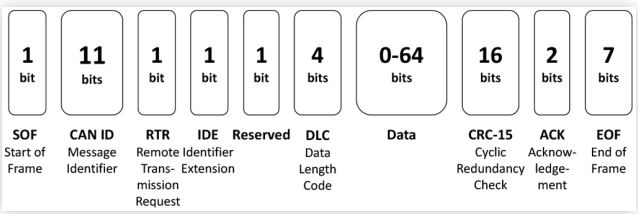


FIGURE 2 IVN Architectures, 7 Layer OSI Model (Adapted from [5])

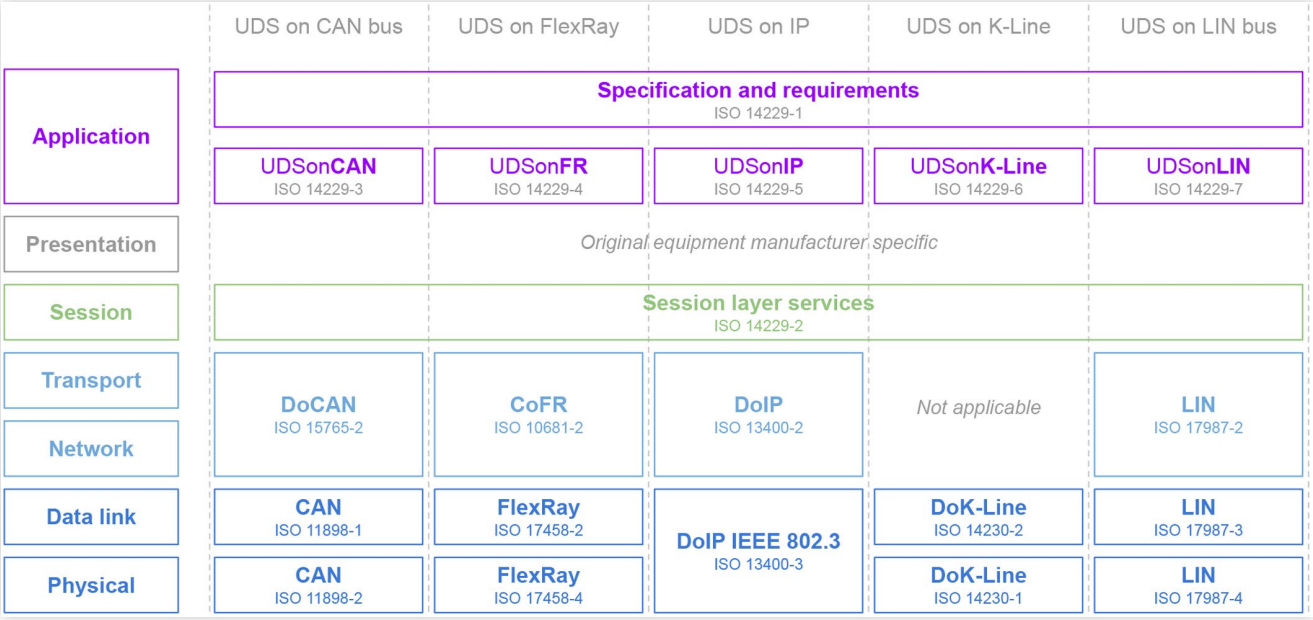


FIGURE 3 UDS request/positive response frame (Adapted from [5])

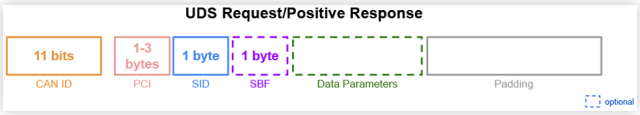
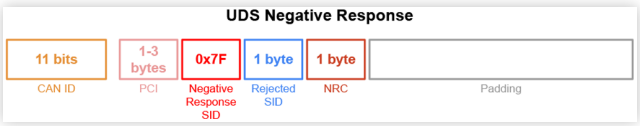


FIGURE 4 UDS negative response frame (Adapted from [5])



- **Rejected SID**: The SID of the corresponding UDS.
- **NRC (Negative Response Code)**: The reason for the rejection.

Fuzz Testing

Fuzz testing involves generating and injecting unexpected inputs into a system to test its robustness. It is a very effective software testing technique to ensure software robustness or uncover unknown vulnerabilities. It is deployed either directly onto the source code or can be run on a network protocol. The typical flow of a fuzzer is as follows:

1. **Send inputs**: Input data to the DUT.
2. **Check response**: Check the response for unexpected behavior.
3. **Instrumentation**: Confirm with an instrumentation message (usually a ping) that DUT still functions.
4. **Repeat**: Modify input every cycle and keep going until a failure or another end condition.

Fuzz Testing Techniques

Various approaches to fuzz testing exist, classified mainly according to the underlying "fuzz engine" – the algorithm to generate fuzz inputs.

Random fuzzing, otherwise known as dumb fuzzing, sends completely random inputs to the target. The fuzz engine has no idea about the structure of the DUT. Moreover, the inputs are not tailored to any specific application.

Popularized by the open-source tool Peach Fuzzer, mutation-based fuzzing starts with a small set of valid inputs and mutates them into a more extensive set with higher case coverage.

Generational fuzzing exploits the protocol structure or the program under test and creates inputs from scratch. The complexity of the protocol is highly related to the effectiveness of generational approaches since a generational fuzz engine can easily follow a complex protocol such as a TLS handshake, where other methods may struggle. However, it needs to be tailored to the protocol or the program. PROTOS [6], Defensics [7], and Peach Fuzzer are well-known fuzzers with generational engines.

One of the challenges with generational fuzzing is the substantial time and expertise required to model the fuzzer according to the specific protocol it is designed to test. The effectiveness of a fuzzer depends heavily on the thoroughness of the software developer who developed

the model, which must be closely aligned with detailed protocol specifications. Especially given that manufacturers may implement standards differently, developing a black-box fuzzer for a protocol is not always feasible. According to program responses, another issue of generational fuzzing is the lack of dynamic test case generation.

Feedback or "on-the-fly" fuzzing is a dynamic approach that uses DUT responses as inputs to the fuzz engine in an online fashion. Unlike traditional methods, which generate all test cases before deployment [8], feedback fuzzing creates inputs during testing, making the testing more directed while ensuring that previous failures are not repeated. As exemplified by American Fuzzy Loop (AFL) [9], feedback fuzzing is frequently employed in practice; however, it is harder to develop compared to traditional methods and is usually deployed directly onto source codes instead of protocols as it may overlook issues specific to protocol implementation.

Related Work

The importance of automotive security was understood in 2015 when an automotive exploit was published in [10], where the researchers used a series of exploits that ultimately allowed them to control a vehicle and steer it off the road remotely. This incident is widely known as the "Jeep Hack", which required a recall of 1.4M vehicles and led to many lawsuits [3]. Given the rapid rise in cyber threats for vehicles, [11] underlines the importance of the automotive sector's adoption of rigorous fuzz testing by demonstrating how fuzzers could be deployed at the UDS layer and criticizes the automotive industry for not having adopted fuzz testing methodology.

Presented at DEF CON 2023, [12] is an LLM tool that automatically generates fuzz tests for relatively small-scale Python code. It demonstrates how LLMs can be very effective at understanding code and generating test cases, which is significant for fuzz testing. Moreover, the tool could try to "auto fix" the code after finding issues [13]. However, it is a white-box source code fuzzer and is not intended to work on protocols.

[14] proposes an LLM-based fuzzer for real-time streaming protocol (RTSP), which can use the LLM as a state machine and can be "guided" online based on the previous responses. The presented benchmark claims that the LLM-based fuzzer has uncovered more vulnerabilities than traditional fuzzers.

Although fuzz testing has been a popular research subject, the literature on CAN fuzzing is limited. [15] outlines creating a UDS fuzzer that uses the PCAN interface to fuzz ECUs. They demonstrate their fuzzer's robustness by causing a software crash on the instrument cluster, the recovery of which required an external power cycle. [16] uses Defensics and Caring Caribou to fuzz open-source UDS implementations and underlines the necessity for open-source security in the automotive industry.

Methodology

Black-Box vs. Gray-Box vs. White-Box Fuzzing

Aside from the input generation algorithms, fuzzers are also grouped into three categories depending on their knowledge of the DUT. The most common is the black-box methodology, where the fuzzer has no internal knowledge about the DUT. The opposite, where the fuzzer has a complete internal understanding of and access to the DUT, is the white-box methodology. A white-box fuzzer is usually designed according to the source code of the DUT and can monitor the DUT with debug access or internal log files [17]. Both approaches have certain advantages; the white-box approach is naturally more powerful and can discover more vulnerabilities in a shorter time frame, while the blackbox approach is transferable between different programs, significantly reducing development costs.

A middle ground that partially benefits from both approaches is the gray-box methodology. A gray-box fuzzer has some internal knowledge of the DUT, usually in a specification, file format, or architecture description. Without a deep understanding of the source code or developer access, this approach is significantly more transferable than the white-box approach. Although limited, the extra knowledge makes it more directed than the black-box approach.

Fuzz Testing Architecture

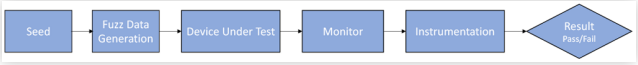
Fuzz testing architecture defines some fundamental principles shared between fuzzers.

The fuzzer's main component is the fuzz engine, which serves as a test case generator.

A "seed" is an initial input to the fuzzer. It can be a value, a file, a network packet, a command-line argument, a set of valid inputs, or any input data the target will accept.

Instrumentation is entering a familiar input, i.e., an input with a known output. Doing so introduces a mechanism to confirm the input that causes a system crash if one occurs. Without instrumentation, the fuzzer cannot distinguish whether the DUT failed to respond due to a system crash caused by the current input, or the previous input despite a delayed response being triggered. Instrumentation can also be used to get additional data from the target to see the state of health of the DUT. Fig. 5 depicts the typical architecture of a fuzz testing tool.

FIGURE 5 Fuzz Testing Architecture



Artificial Intelligence

Recent computational technology and the invention of new machine learning (ML) models have significantly grown the field of AI. A notable example is deep neural network (DNNs), which are based on the biological structure of neurons and consist of multiple "layers" of components [18]. Two recent DNN models are generative adversarial network (GAN) and generative pre-training transformer (GPT); the latter, also called generative AI, is the basis of our work.

The GPT architecture gained significant popularity following OpenAI's release of the ChatGPT in 2020 [19]. GPTs can answer questions, generate creative content, understand various programming languages and code, and even be employed for autonomous driving purposes.

Developing Software with an LLM

GPTs' proficient understanding of short blocks of code and their ability to generate new ones quickly made them an excellent tool for software developers [20]. They have shown that boilerplate code can easily be generated in seconds. Some of the widely accepted "developer companions" are OpenAI's ChatGPT, Microsoft's Github Copilot, and Meta's recent CodeCompose.

Results

Experimental Setup

The DUT is a Model 2022 engine control module (ECM)[1]. The primary purpose of an ECM is to control the engine's fuel injection and spark timing for optimal propulsion. By continuously monitoring and adjusting the engine functions, the ECM ensures efficient combustion, improved performance, and reduced emissions. Fig 6 displays a photograph of the experimental setup.

Experiments were conducted on a virtual Windows 11 ARM (4.5 GB registered RAM, 64-bit) running on an M2 MacBook Pro.

Obtaining Training Data

Defensics was selected as a suitable commercial fuzz testing tool for generating baseline results due to its ability to test protocols at a black-box level and its support for various instrumentation methods.

Defensics was utilized as a benchmarking tool to generate a dataset for evaluating fuzz testing results and for comparison against the AI-generated fuzzer. The AI-generated fuzzer was developed independently and does not rely on Defensics for its functionality. Future work should explore using alternative or independently sourced training data to validate the autonomy of LLM-generated fuzzers fully.

FIGURE 6 Hardware setup for the experiment

TABLE 1 List of UDS services

SID	UDS Service
0x11	ECU Reset
0x28	Communication Control
0x3E	Tester Present
0x85	Control DTC Setting
0x86	Response On Event
0x87	Link Control
0x22	Read Data By Identifier
0x24	Read Scaling Data By Identifier
0x14	Clear Diagnostic Information
0x2F	Input Output Control By Identifier
0x31	Routine Control

A simple probing of the DUT revealed that the UDS services displayed in Table 1 were available on the ECU.

Unrecoverable failures in which the ECU is no longer responsive are possible, which puts the DUT in a "stuck" state. Such failures may present significant security and safety concerns for vehicles since the network containing UDS messages also contains raw, engine-specific CAN data critical for vehicle propulsion.[2] In our experimental setup, we cannot conclusively determine whether the

[1] We will disclose the manufacturer or the exact model of the ECM after the responsible disclosure process has been completed.

[2] The risks of such failures are demonstrated by real-world incidents, such as the 2022 recall by Cummins Inc., where faulty engine control modules (ECMs) caused stalling in over 12,000 engines across various product lines, significantly increasing the risk of crashes and impacting manufacturers like Kenworth and Peterbilt models of Paccar [21].

TABLE 2 CAN bus communication log of a failure.

Entry	Timestamp (DT)	ID	DLC	Data Bytes			
215695	7893163.986	07E0	8	02	10	03	00 00 00 00 00
215696	7893165.992	07E8	8	06	50	03	00 32 01 F4 00
215697	7893186.852	07E0	8	02	11	E0	00 00 00 00 00
215698	7894816.782	07E0	8	02	10	01	00 00 00 00 00
215699	7896332.402	07E0	8	02	10	01	00 00 00 00 00
215700	7899392.766	07E0	8	02	10	01	00 00 00 00 00
215701	7903411.870	07E0	8	02	10	01	00 00 00 00 00
215702	7908430.647	07E0	8	02	10	01	00 00 00 00 00
215703	7908430.935	07E8	8	06	50	01	00 32 01 F4 00

Note: In the table, blue rows indicate healthy communication, gray rows signify instrumentation attempts without response, and the orange row denotes the test case causing the failure.

ECU in the stuck state was completely disabled or only appeared as so due to disconnection from an operational network. However, the ECU entered a bus-off state, effectively removing it from the CAN network. The ECU required a reboot through an external power cycle to recover from this state. We implemented an automated recovery mechanism using a smart plug that reset the ECU's power switch after five consecutive failures.

After statically generating 67,294 test cases, they were sequentially input to the DUT randomly to increase unpredictability, intending to increase code coverage. The total tests took 04:08:14 (HH:MM:SS). The outcome of the tests is presented in Table 4.

Two of the six uncovered vulnerabilities placed the system in the stuck state. A section of the CAN logs corresponding to one such failure is presented in Table 2. The input corresponds to an ECU Reset request with a fuzzed SBF input field.

We presume that either the DUT had faulty software logic that only evaluated the first few bits of the SBF and lacked a robust *else* path for unexpected inputs, or the software (written in C) had encountered memory problems.

We discovered another SBF value that would result in the same failure state. The other four failures were also ECU reset requests processed by the DUT, which put the DUT offline for a relatively long time. Defensics considered this expected behavior since the device transmitted a positive response and processed the ECU reset.

Modeling a Fuzzer with an LLM

Writing a fuzzer targeting a protocol can be challenging, with specifications reaching over a thousand pages. Human interpretation of reading a large specification and software development is often misinterpreted due to a lengthy protocol specification. One way around this issue is to combine an NLP with an LLM to process specifications and generate code accordingly [22]. Creating a protocol corpus using an LLM has shown promise when [14] demonstrated that an LLM could model protocols based on RTSP.

Rather than training a custom LLM, we determined that it would be more effective to use OpenAI's GPT-4 for this study. We first queried GPT-4 with a request to generate Python code based on the UDS specification to see whether it could create a fuzzer based on a standard. We discovered that GPT-4 is aware of many public protocol standards, meaning it would not need to be given the standard explicitly. However, UDS is not a public standard; thus, additional resources must be provided on the protocol. After several prompts, GPT-4 successfully generated a CAN fuzzer with a Python-CAN dependency – an open-source library to provide CAN support for Python. Additionally, we supplied GPT-4 with the CAN log collected through Defensics.

GPT-4 identified the basic structure of the CAN log, including the logged CAN IDs used by UDS, data length codes (DLCs), supported UDS messages sent, and the message length.

GPT-4 had no significant problems creating boilerplate code based on the log and the standard. To enhance the fuzzer, we then asked it to add instrumentation (an essential functionality of fuzzers, as previously discussed) to the boilerplate code so that whether the DUT was still responsive after a fuzz message could be determined. Instrumentation not being part of the original boilerplate underlines that at least some knowledge of the system and fuzz testers is helpful in queries for the resulting program quality.

One of the challenges observed when using an LLM was that the exact wording of prompts dramatically impacted how the LLM would respond. To study how prompts affect the accuracy of an LLM, "prompt engineering" is an emerging field. It suggests various techniques that can be used to achieve better results from

TABLE 3 UDS communication logs as training data

CAN ID	Length Byte	Service ID	Data Bytes	Label
7E0	3	22	D3CC	Pass
7E0	3	22	F213	Pass
7E0	2	27	45	Pass
7E0	2	3E	44	Pass
7E0	2	11	E0	Fail
7E0	3	3E	52C3	Pass

an LLM. One technique worth mentioning is called "think step by step." As the name implies, the user appends the phrase "think step by step" at the end of their prompts. [23, 24] report that this technique significantly improves the accuracy of LLM responses.

Inputting Data to the LLM

The training data of around 67,200 entries after preprocessing is entered into the LLM through a prompt in a "batch prompting" fashion [25], which is the act of feeding the data into an LLM in a single prompt instead of a sequence of prompts. Next, the LLM is prompted to be ready to generate new test cases based on future logs that will be fed into it. This is a **RAG!** (**RAG!**) approach, which combines the strengths of LLMs with a non-parametric, external memory component, typically in the form of a large corpus or database that the model can query to retrieve information. This process enhances the model's ability to generate responses by allowing it to pull in relevant information from an external database in real-time [26].

Deploying the LLM-Generated Fuzzer

We deploy our "LLM UDS Fuzzer" into the same setup as Defensics. We run the fuzzer for the same number of test cases and integrate the same power cycle script.

The LLM UDS Fuzzer reported 46 failures, many of which were repetitions of the same issues due to the lack of a mechanism to avoid previously identified failures. Among these, four were unique failures – the same ones uncovered by Defensics. Table 4 compares the results against Defensics.

Although the LLM-generated fuzzer did not outperform Defensics, we still consider this a successful outcome because Defensics is a commercial tool, and the LLM-generated fuzzer was created almost entirely by an LLM in a relatively short period. We argue that LLM-generated fuzzers can provide a practical alternative to commercial tools when the budget for purchasing such tools is limited, and the time or expertise needed for manually developing a new tool may not be readily available.

Conclusion and Future Work

The primary goal of this paper was to explore whether AI models can speed up the development of fuzz testers. We demonstrated that using an LLM to generate a functional fuzzer based on a protocol specification is possible. We also showed that an LLM can develop a fuzz testing corpus and generate new test cases based on past data, creating a feedback loop that allowed the LLM to utilize **RAG!**. These results suggest that LLMs can provide valuable assistance in fuzz testing.

The study highlighted the potential of LLMs to understand CAN data and create fuzz tests using a feedback fuzzing approach, enabling the generation of directed tests and achieving higher code coverage. However, achieving the same level of thoroughness as a human developer remains an open challenge for AI. LLMs are known to produce fabricated or inaccurate outputs when provided with ambiguous prompts, which poses challenges in generating test cases that fully align with detailed protocol specifications. Future work should investigate methods to enhance the reliability and thoroughness of LLM-generated fuzzers through improved prompt engineering, validation mechanisms, and comparisons with human-developed tools.

Our primary focus was UDS over CAN, but vehicles employ several other protocols. Future efforts will expand this work to include LLM-aided fuzz testing for other vehicle communication buses. Moreover, exploring alternative datasets and testing diverse protocols will be critical to further validating the autonomy and effectiveness of LLM-generated fuzzers.

Despite numerous limitations, we successfully demonstrated a failure in a production ECM, showcasing the potential of LLM-generated fuzzers as complementary tools in automotive fuzz testing. While ISO/SAE 21434 lists fuzz testing in their recommended testing techniques for the automotive industry, it stops short of making it a requirement [27]. We argue for the necessity of industry-wide regulations to mandate standardized fuzz testing to ensure higher safety and security standards.

Discussion

Overview

The necessity of fuzz testing, a critical technique for identifying vulnerabilities in many fields, was discussed, and its seeming lack of adoption in the automotive field was underlined. The impact of LLMs on fuzz testing was explored, both in test-case generation and fuzzer program development. A fuzzer capable of fuzzing real hardware was developed entirely through prompts to GPT-4, which proved to be able to find a potentially safety-critical vulnerability in an ECU from modern vehicles. Although

TABLE 4 Fuzz test results

Fuzzer	Total Tests	Passed Tests	Failed Tests	Unique Failures	Runtime	Failure Rate
Defensics	67293	67287	6	4	04:18:44	0.0089%
LLM UDS Fuzzer	67293	67247	46	4	10:01:33	0.0684%

we demonstrate the potential of LLMs to generate fuzzer programs, there remains significant room for improvement.

Limitations

Testing automotive ECUs outside of a vehicle can be challenging because they require a wiring harness for communication, making testing multiple ECUs both costly and labor-intensive. Therefore, our study focuses on testing a single ECU. While testing multiple devices with various tools could produce a broader range of data and help mitigate issues like overfitting, this was beyond the scope of our current work.

Additionally, we limited our focus to the relatively simple UDS protocol, so we refrain from making generalized claims about more complex protocols.

Using a variety of LLMs and comparing the resulting programs could lead to further conclusions; however, we were limited to using only GPT-4. Consequently, our findings are limited to demonstrating that LLMs can develop fuzzers with relative ease compared to manual development without making broader performance comparisons.

A notable limitation of this study was the use of Defensics-generated data to help provide context to the LLM. While the AI-generated fuzzer showed the ability to generate test cases autonomously, it is unclear whether it could achieve the same performance without using commercially generated training data, raising questions about the LLM-generated fuzzer's broader applicability in scenarios where such data is unavailable.

Our future work will include the use of other AI models, as well as the training of one specific to our use case. In addition, we plan to test the autonomy and transferability of LLM fuzzers through more experiments.

References

1. Luo F., Zhang X., Yang Z., et al., "Cybersecurity Testing for Automotive Domain: A Survey," *Sensors (Basel, Switzerland)*, vol. 22, no. 23, p. 9211, Nov. 2022, doi: 10.3390/s22239211. [Online]. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9736493/ (visited on 11/19/2023).
2. *Road vehicles — Unified diagnostic services (UDS) — Part 1: Application layer* (Geneva, CH, Standard: International Organization for Standardization, Feb, 2020)
3. Pesé, M.D., "Bringing practical security to vehicles," PhD dissertation, The University of Michigan, 2022.
4. *Road vehicles - diagnostic communication over controller area network (docan) - part 2: Transport protocol and network layer services* (Geneva, CH, Standard: International Organization for Standardization, Apr 2016)
5. Falch, M., Uds explained - a simple intro, 2022, accessed: October 9, 2023, CSS Electronics, https://www.csselectronics.com/pages/uds-protocol-tutorial-unified-diagnostic-services.
6. Kaksonen, R., "A Functional Method for Assessing Protocol Implementation Security," PhD dissertation, Jan. 2001.
7. Defensics, https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html, accessed: 2023-10-26, Synopsys Inc., 2023.
8. Fioraldi, A., D'Elia, D.C., and Balzarotti, D., "The use of likely invariants as feedback for fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, 2829-2846, isbn: 978-1-939133-24-3. [Online]. https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi.
9. Zalewski, M., American fuzzy loop, 2013, accessed: 2023-10-11, https://github.com/google/AFL.
10. Miller, D.C. and Valasek, C., "Remote Exploitation of an Unaltered Passenger Vehicle," en, Aug. 10, 2015.
11. Bayer, S., and Ptok, A., "Don't Fuss about Fuzzing: Fuzzing Controllers in Vehicular Networks," 2015. [Online]. https://www.escar.info/images/Datastore/2015_escar_EU_Papers/3_escar_2015_Stephanie_Bayer.pdf.
12. Xavier, LLM Fuzz (previously fuzzforest), original-date: 2023-03-09T16:42:54Z, Nov. 2023. [Online]. https://github.com/tree-wizard/fuzz-forest (visited on 11/19/2023).
13. Cadena, X., LLMs to Write Fuzzers - Infinite Forest, en. [Online]. https://infiniteforest.org/LLMs+to+Write+Fuzzers (visited on 11/19/2023).
14. Meng, R., Mirchev, M., Bohme, M., and Roychoudhury, A., "Large Language Model guided Protocol Fuzzing," in *Proceedings of the Network and Distributed System Security (NDSS) Symposium 2024*, San Diego, CA, USA: The Internet Society, Feb. 2024, isbn: 1-891562-93-2, doi: 10.14722/ndss.2024.24556. [Online]. https://dx.doi.org/10.14722/ndss.2024.24556.
15. 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)*, Luxembourg: IEEE, Jun. 2018, 239-246, isbn: 978-1-5386-6553-4, doi: 10.1109/DSN-W.2018.00070. [Online]. https://ieeexplore.ieee.org/document/8416255/ (visited on 11/18/2023).
16. 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.
17. Kreissl, J., Fuzzing techniques - The Generator Menace, en-US, Feb. 2021, [Online]. https://www.coderskitchen.com/fuzzing-techniques/ (visited on 11/21/2023).
18. Yi, H., Shiyu, S., Xiusheng, D., and Zhigang, C., "A study on Deep Neural Networks framework," in *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, Xi'an, China: IEEE, Oct. 2016, 1519-1522, isbn: 978-1-4673-9613-4, doi: 10.1109/IMCEC.2016.7867471. [Online]. http://ieeexplore.ieee.org/document/7867471/ (visited on 10/20/2023).
19. Yenduri, G., Ramalingam M., Chemmalar Selvi G., Supriya Y., et al., Generative Pre-trained Transformer: A

Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions, arXiv:2305.10435 [cs], May 2023, [Online]. http://arxiv.org/abs/2305.10435 (visited on 10/23/2023).

20. Hou, X., Zhao, Y., Liu, Y., et al., Large Language Models for Software Engineering: A Systematic Literature Review, arXiv:2308.10620 [cs], Sep. 2023, [Online]. http://arxiv.org/abs/2308.10620 (visited on 11/07/2023).
21. C. Inc. "Cummins recalls over 12,000 engines due to faulty engine control modules (ecms)," Accessed: 2024-12-18. (Aug. 2022, [Online]. https://www.freightwaves.com/news/cummins-recalls-faulty-engine-control-modules-from-28-manufacturers.
22. Xia, C.S., Paltenghi, M., Tian, J.L., Pradel, M., and Zhang, L., Universal Fuzzing via Large Language Models, arXiv:2308.04748 [cs], Aug. 2023. [Online]. http://arxiv.org/abs/2308.04748 (visited on 11/05/2023).
23. Gao, A., Prompt Engineering for Large Language Models, en, SSRN Scholarly Paper, Rochester, NY, Jul. 2023. doi: 10.2139/ssrn.4504303, [Online]. https://papers.ssrn.com/abstract=4504303. (visited on 11/14/2023).
24. Kojima, T., Gu, S.S., Reid, M., Matsuo, Y., and Iwasawa, Y., Large Language Models are Zero-Shot Reasoners, arXiv:2205.11916 [cs] version: 4, Jan. 2023, [Online]. http://arxiv.org/abs/2205.11916 (visited on 11/15/2023).
25. Zhang, H., Dong, Y., Xiao, C., and Oyamada, M., Large Language Models as Data Preprocessors, arXiv:2308.16361 [cs], Aug. 2023, [Online]. http://arxiv.org/abs/2308.16361 (visited on 11/17/2023).
26. Lewis, P., Perez, E., Piktus, A., et al., Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks, arXiv:2005.11401 [cs], Apr. 2021, [Online]. http://arxiv.org/abs/2005.11401 (visited on 11/17/2023).
27. "Road vehicles - cybersecurity engineering," International Organization for Standardization and Society of Automotive Engineers International, Geneva, CH, Standard, Aug. 2021.

Contact Information

John McShane
Eastern Michigan University
jmcshane@emich.edu

Iwinosa Aideyan
Clemson University
iaideya@clemson.edu

Mert D. Pesé
Clemson University
mpese@clemson.edu

Levent Çelik
Clemson University
lcelik@clemson.edu

Richard Brooks
Clemson University
rrb@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).

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited. OPSEC # 9004

Definitions, Acronyms, Abbreviations

AFL - American Fuzzy Loop
CAN - Controller Area Network
CRC - Cyclic Redundancy Check
DLC - Data Length Code
DNN - Deep Neural Network
DUT - Device Under Test
ECM - Engine Control Module
ECU - Electronic Control Unit
GAN - Generative Adversarial Network
GenAI - Generative AI
GPT - Generative Pre-training Transformer
IC - Intellectual Contribution
IVN - In-Vehicle Network
LLM - Large Language Model
ML - Machine Learning
OEM - Original Equipment Manufacturer
RTSP - Real-Time Streaming Protocol
UDS - Unified Diagnostic Services

ISSN 0148-7191