2024-01-2795 Published 09 Apr 2024

Fuzzing CAN vs. ROS: An Analysis of Single-Component vs. Dual-Component Fuzzing of Automotive Systems

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

Citation: Aideyan, I. W., Brooks, R., and Pese, M.D.,"Fuzzing CAN vs. ROS: An Analysis of Single-Component vs. Dual-Component Fuzzing of Automotive Systems," SAE Technical Paper 2024-01-2795, 2024, doi:10.4271/2024-01-2795.

Received: 20 Oct 2023 Revised: 29 Jan 2024 Accepted: 05 Feb 2024

Abstract

Robust communications are crucial for autonomous military fleets. Ground vehicles function as mobile local area networks utilizing Controller Area Network (CAN) backbones. Fleet coordination between autonomous platforms relies on the Robot Operating System (ROS) publish/subscribe robotic middleware for effective operation. To bridge communications between the CAN and ROS network segments, the CAN2ROS bridge software supports bidirectional data flow with message mapping and node translation.

Fuzzing, a software testing technique, involves injecting randomized data inputs into the target system. This method plays a pivotal role in identifying vulnerabilities. It has proven effective in discovering vulnerabilities in online systems, such as the integrated CAN/ROS system. In our study, we consider ROS implementing zero-trust access control policies, running on a Gazebo test-bed connected to a CAN bus. Our objective is to evaluate system security using fuzzers in three scenarios: (i) fuzzing the CAN bus alone, (ii) fuzzing the CAN bus with a ROS Fuzzer, and (iii) fuzzing both systems simultaneously using the CAN2ROS bridge.

This paper poses the question: is fuzzing the unified system more effective than fuzzing individual components. By analyzing interactions between the bridge and the military fleets' CAN systems, we identify and address flaws potentially introduced in the software, or data leakage between communication segments. Our analysis employs experimental design and statistical analysis to shed light on the bridge's security robustness and its potential implications for the overall system's integrity.

This research holds significant implications for both industry and academia. Stakeholders involved in the development of autonomous military and civilian fleets can leverage our findings to enhance system security and reliability. Ultimately, the identification and mitigation of vulnerabilities contribute to safer and more resilient military operations.

Introduction

In the domain of autonomous military fleets and unmanned systems, maintaining robust and resilient communications within vehicles is vital for their safe and effective operation. This paper delves into the concept of threat modelling and analyzing a vehicle's attack surface to uncover potential attack possibilities. Attack paths are identified from the attack surface, describing routes an attacker may take from a point of entry to target components, highlighting potential points of exploitation [1]. Recent developments in automated red teaming have seen fuzzing techniques emerge as powerful tools for detecting and preventing security flaws across various attack surfaces[1]. Fuzzing, a dynamic and automated software testing technique, plays a crucial role in discovering vulnerabilities and security weaknesses in target systems. The primary objective of fuzzing in this context is to identify potential points of exploitation, such as software vulnerabilities, which malicious actors could leverage to gain unauthorized access or compromise the target system. Autonomous military fleets and unmanned systems utilize diverse communication technologies, including Controller Area Network (CAN), Local Interconnect Network (LIN), Automotive Ethernet, FlexRay, and ROS-Military. Existing literature has shed light on the fundamental aspects of vehicle network communication; however, it has not fully explored the possible connection from the robotic middle ware (ROS) of autonomous ground vehicles to the CAN bus. The CAN bus serves as an essential communication backbone enabling various Electronic Control Units (ECUs) to share sensor data, control commands, and other vital information to coordinate the vehicle's autonomous operation. This connection presents a potential attack surface that needs careful consideration. This research aims to address this gap by focusing on the automated discovery of security flaws in vehicle fleets, particularly investigating an integrated CAN-to-ROS system. We employ fuzzing techniques to evaluate the robustness and security of both the CAN

bus and ROS network, with access control policies considered for added security measures.

The paper structure is as follows: Section 2 provides an overview of existing work and related technologies, including CAN, ROS, Fuzzing, and the CAN-to-ROS bridge. Section 3 delves into the overall network model and design choices for implementing test-beds for autonomous fleets, covering aspects such as CAN bus and ROS1 Gazebo test beds, data transmission, and the mapping algorithm translator node. Section 4 evaluates the performance of the proposed approach, presenting experimental results. We conclude with section 5, a discussion of the findings takes place, and the paper concludes by summarizing the results.

Background

Autonomous military fleets and unmanned systems are revolutionizing modern military operations, with applications spanning reconnaissance, surveillance, logistics, and more. These technologies enhance operational efficiency and reduce personnel risk by eliminating human presence in hazardous or challenging environments. However, the use of these systems in combat roles is a subject of ongoing debate, particularly in light of US doctrine, which emphasizes human decision-making in the use of lethal force. This aspect remains a controversial and ethically complex area in the evolution of military technology [2]. Autonomous military fleets, comprising interconnected vehicles, leverage advanced sensors, AI algorithms, and communication protocols to enable coordinated and synchronized actions and, it is crucial to have a comprehensive understanding of threat modeling. The latter is a structured approach that evaluate vulnerabilities of a vehicle's attack surface. By identifying and analyzing attack paths, threat modeling helps uncover how malicious actors could exploit the system. This analysis traces an attacker's trajectory from their initial point of entry into the system, through internal networks, and ultimately to specific target components. By conducting this intricate analysis, valuable insights can be gained to fortify the system against emerging threats [3].

In-Vehicle Network Communication Technologies

In military autonomous ground vehicles, specialized internal communication networks serves as the backbone that interconnects components within the vehicle. These networks are designed to meet demands, such as the guaranteed delivery of messages and the prevention of message conflicts. Vehicle network communication technologies used by military autonomous ground vehicles include standards such as the Controller Area Network (CAN), Local Interconnect Network (LIN), Automotive Ethernet, and FlexRay. These technologies provide robust and secure communication across the intricate landscape of a military autonomous vehicle. The Robot Operating System (ROS) adds to vehicles' communication capabilities.

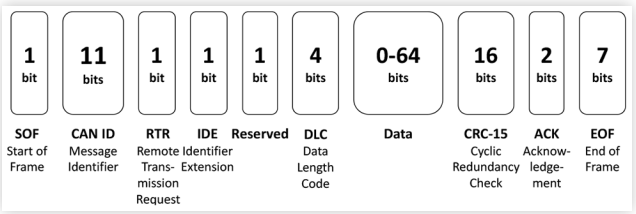
Controller Area Network

The Controller Area Network (CAN) is a widely adopted communication protocol designed for robust and reliable data exchange between electronic components in vehicles and industrial systems. CAN operates on a message-based communication model, where devices connected to the network transmit and receive messages, referred to as "CAN frames. The frames include the message identifier (ID), data payload, and control bits, which allow vehicle to distinguish and prioritize messages. This protocol's efficacy stems from its capacity to support multiple devices on the same network without a centralized controller. CAN and its associated frame structure are part of modern autonomous vehicle networks, fostering inter-component communication critical for operation of these advanced vehicles.

Figure 1 depicts a CAN data frame, which is used for reliable data transmission between nodes on a CAN bus. The respective fields are explained as follows:

- **SOF** (Start of Frame): This is a 1-bit field that marks the beginning of a new CAN frame, indicating the start of data transmission on the network.
- **CAN ID**: This field uses the CAN 2.0A standard 11-bit (or 29-bit in CAN2.0B) message identifier for arbitration, setting the priority of the data frame and it allows for up to 2048 different CAN IDs.
- **RTR** (Remote Transmission Request): This 1-bit field distinguishes between data frames and remote frames, indicating whether it is a data frame or a request for data from other nodes.
- **Control**: This field typically contains defined bits and additional control information, including the IDE (Identifier Extension) bit. This bit, when dominant, set to 0, defines the 11-bit standard identifier, while when recessive, defines the 29-bit extended identifier. It indicates the identifier format being used.
- **DLC** (Data Length Code): DLC stands for Data Length Code and is 4 bits in length. It defines the data length in the data field, indicating how many bytes of data are present.

FIGURE 1 CAN Message Format



- **Data**: The data field can contain up to 8 bytes of actual data, which are transmitted between nodes on the CAN bus.
- **CRC** (Cyclic Redundancy Check) Field: This 15-bit field is used to detect any corruption during data transmission, helping ensure data integrity.
- **ACK** (Acknowledgment) Field: In CAN, acknowledgment is handled within the frame itself. Successful receipt and validation of the data frame result in an ACK bit being sent to acknowledge receipt, while the absence of an ACK indicates an error. Wired-AND logic requires at least one receiver to transmit a 0 to acknowledge CAN frame receipt, and if no receiver acknowledges the frame, the transmitter re transmits it.
- **EOF** (End of Frame): This field consists of 7 consecutive recessive bits known as "End of Frame", marking the end of the CAN frame and preparing the network for the next frame.

Robot Operating System

The Robot Operating System (ROS) serves as a middleware that enables coordination between robotic systems and vehicle components. ROS is an open-source operating system for robotics that provides tools and libraries for creating applications. It runs as nodes, which communicate through a publisher/subscriber scheme. This scheme handling low-level interactions. By providing a framework for modular software development, ROS facilitates the creation of autonomous systems.

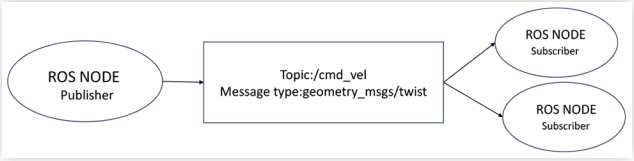
ROS provides services, such as communication among processes, hardware abstraction, low-level device control, and package management, for a heterogeneous computer cluster. Nodes in a graph architecture, conduct processing and may receive, publish, and manipulate perception, control, and other messages, as in Figure 2. Although not a real-time operating system, it is possible to integrate ROS with real-time code. Specifically, ROS1, the earlier version of ROS, introduced tools and libraries for robotic applications, fostering collaborative development within the robotics community. ROS2, the successor to ROS1, addresses its limitations and enhances real-time performance, security, and additional platform support [4].

ROS uses a standardized message format to facilitate communication between nodes (software modules). The ROS message format is flexible and extensible. The syntax is:

- **Message Name**: Each ROS message has a name that describes its purpose or content.
- **Message Type**: ROS messages are strongly typed, meaning they have a specific data type associated with them. Common data types include integers, floats, strings, arrays, and custom user-defined types.
- **Message Fields**: Messages consist of one or more fields, each with a name and a data type. These fields store the actual information being communicated. Let us assume that these fields are responsible for storing the velocity information necessary to control the robotic system's motion. For instance, a custom message contains the following fields:

  linear float32 x float32 y float32 z
  Angular float32 x float32 y float32 z
- **Header**: Many ROS messages include a standard "header" field that contains metadata such as a timestamp and frame ID. This header is useful for tracking when the message was generated and in which coordinate frame it applies.

FIGURE 2 ROS Publisher and Subscriber



Vehicle Hardware Abstraction Layer

The Vehicle Hardware Abstraction Layer (VHAL) serves as an intermediary layer that abstracts and standardizes the interaction between various hardware components, sensors, and control systems in a vehicle. Essentially, VHAL acts as a unifying interface, providing a consistent and high-level opportunity for software applications and vehicle control systems to communicate with and control the diverse hardware elements in an automobile. This abstraction layer streamlines software development, testing, and integration processes, fostering compatibility and interoperability across different vehicle platforms and facilitating advanced automotive research endeavors. It acts like a bridge that helps different parts of a car's computer systems communicate and work together.

The `socketcan bridge` package in ROS uses the VHAL concept. It serves as a bridge between the ROS ecosystem and the CAN bus, allowing for communication and control of the vehicle's hardware components. While it may not encompass all aspects of VHAL, it abstracts and standardizes the interaction between ROS-based software and the CAN bus hardware, helping facilitate communication, data exchange, and control between these two systems in the context of a vehicle.

The `socketcan bridge` and ROS-to-CAN Translator play pivotal roles in bridging the communication gap between the CAN protocol and the ROS environment. The `socketcan bridge` serves as a vital link by enabling direct communication between the CAN hardware and ROS nodes. On the other hand, the ROS-to-CAN Translator node assumes the role of a translator, facilitating bidirectional data flow between the CAN and ROS worlds. This bridge is instrumental in mapping CAN messages to ROS message types and vice versa, allowing seamless information exchange and synchronization. The utilization of

these packages effectively empowers ROS-enabled autonomous vehicles to harness the capabilities of CAN communication within the ROS ecosystem.

Importance of CAN-to-ROS Integration

The integration of the Controller Area Network (CAN) bus and the Robot Operating System (ROS) holds significant importance in autonomous military fleets and unmanned systems. This fusion of communication frameworks bridges the gap between hardware-level communication of the CAN bus and higher-level functionalities facilitated by ROS, unlocking numerous advantages while presenting challenges. The integration facilitates seamless data exchange, robust control and coordination, enhanced perception and decision-making, modularity and scalability, unified communication, modularity and reusability, and sensor fusion and decision making.

By integrating CAN bus with ROS, the communication landscape is unified, streamlining data flow across sensors, actuators, and control units, enhancing vehicle efficiency and responsiveness. ROS offers a modular architecture for software development, allowing developers to create ROS nodes that communicate seamlessly with CAN-based hardware, enhancing code reusability and accelerating the development of new functionalities critical in dynamic military operations. Overall, the integration of CAN bus and ROS can enhance the overall vehicle's autonomy and safety, ensuring a seamless and efficient communication system for autonomous military fleets. The synergistic integration of the Robot Operating System (ROS) with the structured CAN frame framework yields a powerful combination that greatly enhances the capabilities of vehicle networks in the context of autonomous operations. This integration facilitates the seamless coordination of diverse components within the vehicle, culminating in the realization of efficient and sophisticated autonomous systems.

Fuzzing and its role in security testing

Fuzzing has been a popular technique for discovering software security vulnerabilities since the early 1990s. It involves repeatedly running a program with generated inputs that may be syntactically or semantically malformed. Fuzz testing is a software testing technique that uses fuzzing to find security-related bugs in a program under test (PUT), including program crashes [5]. It has a specific goal of finding bugs that violate a specific security policy, such as program crashes.

The taxonomy of fuzzers is divided into three groups based on the granularity of semantics a fuzzer observes in each fuzz run. These groups are called black-, grey-, and white-box fuzzers. Black-box fuzzers are techniques that do not see the internals of the PUT, treating it as a black-box. White-box fuzzers generate test cases by analyzing the internals of the PUT and the information gathered when executing the PUT [6]. Additionally, fuzzing can be performed at different levels of the software stack, from the application layer down to the network protocols, providing a comprehensive assessment of the system's security and reliability.

Related Work

Elmadani *et al.* [7] focus on decoding CAN bus messages and the development of a ROS-based package (CAN-to-ROS) for real-time and offline monitoring and decoding showcases a crucial step toward harnessing the wealth of information present in CAN messages. By leveraging the strengths of the Robot Operating System (ROS), the paper effectively addresses the need for modularity, ease of integration, and diverse language compatibility. The approach of integrating the CAN-to-ROS package with the libpanda library facilitates real-time data access and decoding presents a practical solution for working with CAN bus data from actual vehicles. Moreover, the successful evaluation and testing of the package on a Raspberry Pi using real CAN bus data from a Toyota RAV4 shows its viability for real-world scenarios. Specifically, providing additional insights into the security considerations and potential challenges when interfacing with vehicle systems, especially in the context of autonomous military fleets and security-critical applications. Addressing security considerations and scalability aspects could further strengthen the paper's applicability. Nice *et al.* [8] leveraged CAN data within ROS to enhance vehicular control by introducing CAN Coach, a ROS node that interfaces with a CAN data to provide real-time feedback to human drivers. Experiments were conducted on a freeway route involving two vehicles: a lead vehicle and an ego vehicle to test various control objectives, including maintaining a Constant Time-gap, Velocity Matching, and Dynamic Time-gap control. The results demonstrated that providing feedback to human drivers through CAN data, integrated with ROS, can contribute to improved traffic flow and safety. The paper highlights the need for measurements of vehicle velocity, relative velocity, and space-gap (distance to the lead vehicle). This data is obtained from radar sensors and wheel encoders connected to the CAN bus of a 2020 Toyota Rav4 Hybrid with Adaptive Cruise Control (ACC) and a data logging device (Gray Panda) connected to a Raspberry Pi 4. It provides insights into CAN Data Handling (how CAN data can be accessed and utilized within the ROS framework) and use of ROS for real-time data processing, which was essential for this study.

In automotive cybersecurity, research has extensively unveiled vulnerabilities in vehicle network communication [9], emphasizing the imperative for secure communication within contemporary vehicles. Recognized standards like SAE J3061**saej3061** and ISO 26262 **iso26262**address these cybersecurity concerns in the automotive industry. While traditional approaches, exemplified by penetration

testing, have proven effective, they grapple with challenges of being time-consuming, costly, and sometimes hazardous. Pradeep *et al.*[10] contribute significantly by presenting a hardware-in-loop-based automotive cybersecurity evaluation testbed. This testbed endeavors to establish a secure virtual environment for evaluating cyber threats and risks associated with vehicular attacks. The framework integrates vehicle and powertrain models, mobility and network simulators, and actual hardware running control algorithms using CAN communication, focusing on expedited testing and assessment within a simulated environment. While the referenced paper predominantly addresses cybersecurity through simulating attacks in a controlled setting, our work extends into the performance-oriented dimension of automotive cybersecurity, connecting CAN communication to the ROS environment and scrutinizing that intersection.

The study by Liu *et al.*[11] investigates optimal data transmission for collaborative driving in autonomous vehicular networks (AVNs) using the Robot Operating System (ROS) and addresses challenges in scheduling contending data flows in bandwidth-constrained vehicular networks. It models the ROS-based scheme as an optimization problem, using Lyapunov Optimization to calculate power allocation and conflict avoidance. The contributions include multi-user collaboration, joint performance optimization, and ensuring system stability over a long time span. However, this paper notably omits an in-depth exploration of the data communication intricacies between ROS and the Controller Area Network (CAN), a critical aspect for the seamless operation of autonomous vehicular networks. Understanding the dynamics of data exchange between these systems is crucial for achieving comprehensive optimization in autonomous vehicular networks.

Methodology

In this work, message-based fuzzing, a form of white-box fuzzing, was used. It is also known as protocol or network message fuzzing and is a technique used in software testing and security assessment to discover vulnerabilities and defects in network protocols, communication interfaces, and message-driven systems. It can be valuable for uncovering issues such as input validation vulnerabilities, message handling errors, and buffer overflows that may not be apparent during regular operation, as well as other software defects that could be exploited by attackers. In the following, we describe how message-based fuzzing works:

- **Target**: Typically applied to communication protocols, network services, or any software that processes incoming messages or data packets, such as network servers, web applications, IoT devices, and more.
- **Test Input Generation**: Fuzzing tools generate a wide range of test messages, data packets, or inputs that conform to the protocol or data format being tested. These test inputs often include valid messages as well as random (it generates random or semi-random data as test inputs, exploring a wide input space), intentionally malformed, or mutated ones (it starts with valid inputs and applies mutations, such as bit flips, additions, deletions, or permutations, to create variations).
- **Delivery to Target**: The test messages that have been generated are transmitted to the target software or system via a network connection or through suitable communication channels, such as the CAN bus or ROS topics.
- **Monitoring and Analysis**: Fuzzing tools consistently observe the actions and responses of the target system while it is engaged in processing the test messages. The observers monitor for any unforeseen or atypical reactions, such as system crashes, freezes, error messages, memory inefficiencies, or any other abnormal occurrences.
- **Bug Identification**: When the fuzzing tool identifies an inconsistency or atypical behavior, it notifies this occurrence as a prospective vulnerability or flaw. Subsequently, security analysts or developers may proceed to examine the matter in order to ascertain its level of seriousness and potential consequences. In this study, defects was deliberately introduced into the source code of a CAN simulation and exploited a vulnerability in the access policy code that shields the ROS environment against external IP addresses not registered on the network. The objective was to induce unusual behavior and examine the system's response. The crash time of the application running on the different platforms was recorded.

In a system, such a a s ROS or a CAN bus network, transmitting random data to a specific topic or employing random CAN IDs and payloads is as a variant of message-based fuzzing. This is how it is applicable to each of these situations:

ROS Topic Fuzzing The act of transmitting arbitrary or unexpected data to a designated ROS topic can be understood as a means of subjecting the message processing capabilities of the ROS nodes that are subscribed to said topic to a form of fuzzing. The objective is to analyze the behavior of the ROS system when it encounters messages that are unexpected or malformed. This practice can facilitate the identification of vulnerabilities or deficiencies in the logic employed by ROS nodes for message handling.

CAN Bus Fuzzing The act of transmitting random CAN Identifiers (IDs) and payloads can be classified as a demonstration of CAN bus fuzzing. The latter is a technique that entails the transmission of malformed or unanticipated messages over a Controller Area Network (CAN). The primary objective of this practice is to assess the resilience and security of the interconnected systems reliant on the CAN bus. The user's text does not provide any information. The manipulation of CAN IDs and

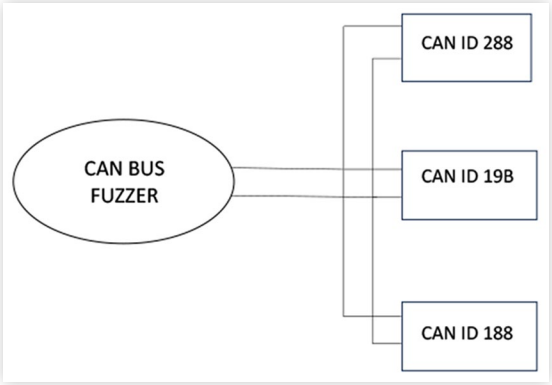
payloads enables the examination of the response of various nodes within the CAN bus to unanticipated or invalid messages. This process has the potential to unveil vulnerabilities or deficiencies in the communication protocols employed by the network.

Experimental Setup

In the CAN segment, we employed three Raspberry Pi units was used and equipped with the PICAN Shield, transforming them into distinct Electronic Control Units (ECUs) simulating critical vehicle functions: the Speedometer ECU (CAN ID: 244) monitors and controls speedometer functionality, ensuring real-time representation of the vehicle's speed; the Door Lock ECU (CAN ID: 19B) manages the vehicle's door locking system, enhancing security and convenience; the Pointer Signal ECU (CAN ID: 188) oversees various instrument cluster indicators, interpreting signals from sensors and systems to provide essential visual feedback to the driver. To evaluate the resilience of these Raspberry Pi-based ECUs, they were subjected to three types of bug namely dereferencing pointer (A), buffer overflow(B), and divide by zero (D) assessing their robustness and dependability under challenging conditions.

The CAN-to-ROS integration in the ROS segment is dependent on three essential nodes: the socketcan bridge, which facilitates communication between ROS and the CAN bus; the socketcan interface, which offers a reliable access to the SocketCAN driver; and the ros to can node. By implementing the ROS Noetic framework, Gazebo [12] was employed. It is a robotics simulation software seamlessly integrated with ROS to serve as a robust platform for constructing a dynamic simulation environment, orchestrating the autonomous movements of three Husky robots. This virtual setting proved instrumental in the testing and refinement of the CAN-to-ROS integration, Within this simulated domain, the Husky robots were meticulously configured to subscribe to the cmd vel topic for precise movement control, with the ROS master facilitating seamless communication among the Huskies and other integral system packages

FIGURE 3 CAN Bus Setup

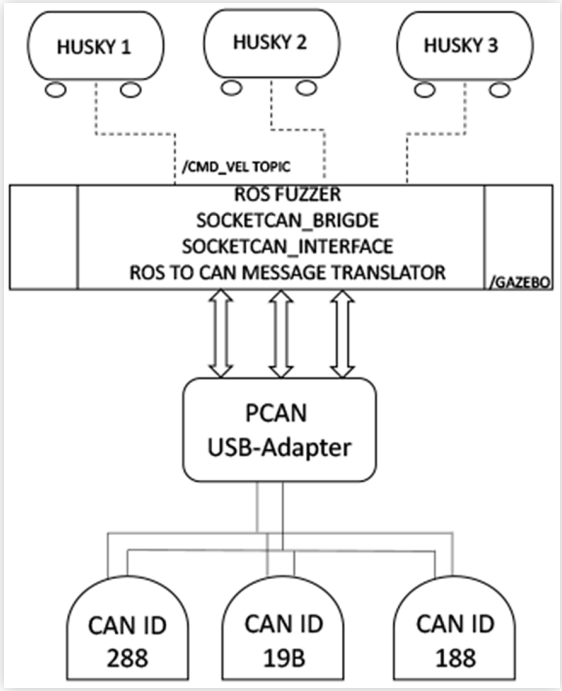


To provide smooth communications between the CAN and ROS components, the socketcan bridge package was utilized. The aforementioned package facilitates the transmission of Controller Area Network (CAN) frames from the socketCAN interface to a topic in the Robot Operating System (ROS). It achieves this by exploiting the socketcan interface provided by the ros canopen package. The integration facilitated the utilization of both conventional and extended CAN frames. The socketcan interface package introduced three crucial functionalities, namely StateInterface, CommInterface, and DriverInterface. These functionalities encompass monitoring the state of the driver, handling message receiving and transmission, and integrating vital management capabilities.

In order to construct the bridge between the Controller Area Network (CAN) and the Robot Operating System (ROS), we employed the Peak CAN USB adapter[13]. This adapter facilitated a simplified connection between CAN networks and the ROS network. The bridge incorporated a node within the /gazebo package, which facilitated message-level communication between the CAN and ROS systems. The aforementioned node performs the task of converting data obtained from the ROS topic /cmd vel into the appropriate format for CAN messages, which are subsequently transmitted over the CAN bus. The conversion and transmission process were managed by the socketcan bridge node, which published received frames on the received messages topic and transmitted messages to the SocketCAN device.

During the testing scenario, the ROS fuzzer was responsible for generating geometric messages on the / cmd vel topic. The ROS-to-CAN node facilitated the conversion of these messages into the CAN format, enabling the subsequent application of fuzzing techniques

FIGURE 4 Integrated Network



on the CAN bus. The extensive configuration facilitated a full evaluation of the CAN-ROS system, leading to the identification of software crashes on each Raspberry Pi. This analysis yielded significant findings regarding the system's resilience and dependability.

Fuzzing process

Following a comprehensive analysis of three prominent open-source fuzzing tools, namely Linux CAN Utils (Cangen)[14], Caring Caribou Fuzzer[15], and SavvyCAN[16], it became evident that the Caring Caribou Fuzzer Module outperformed its counterparts in two critical aspects, namely the generation of valid messages and the ability to induce crashes within the CAN bus network. Notably, it exhibited superior performance both in scenarios with and without additional network traffic. Consequently, the Caring Caribou Fuzzer Module was selected as the primary candidate for further scrutiny and evaluation within the integrated system. Renowned as a dedicated automotive security exploration tool for the CAN bus, its track record of effectiveness made it the logical choice for CAN fuzzing in our study.

In parallel, a custom ROS fuzzer was crafted to fulfill specific requirements. Designed to generate test cases within the system. Its objective was to ensure the creation of valid messages and provoke crashes within the CAN bus network, both under normal operating conditions and scenarios with heightened network activity. The choice to proceed with the ROS fuzzer for subsequent testing and analysis of the integrated system rested on its robust performance in meeting the following essential criteria:

1. **Fuzzing the CAN Bus Alone**: This scenario centers on the standalone fuzzing of the CAN bus. Utilizing the Caring Caribou Fuzzer Module, random fuzzing is executed across all possible CAN IDs within the network until a bug is triggered. The elapsed time taken to induce a crash is automatically recorded by a dedicated Bash script, providing insights into the efficiency and effectiveness of the fuzzing process.
2. **Fuzzing the CAN Bus with a ROS Fuzzer**: In this scenario, the ROS environment is initiated using roslaunch. Subsequently, the `socketcan bridge` node and ros to can node are activated, followed by the execution of the ROS fuzzer node. The core premise involves employing the ros to can node to actively translate ROS messages into CAN bus-compatible formats for the purpose of fuzzing. This methodology assesses the interplay and compatibility between ROS and the CAN bus during fuzzing operations.
3. **Fuzzing Both Systems Simultaneously Using the CAN-to-ROS Bridge**: This progressive scenario entails the concurrent deployment of both the CAN Fuzzer and ROS Fuzzer to identify vulnerabilities and provoke crashes. The duration required to destabilize the program is meticulously recorded, offering insights into the overall robustness and resilience of the integrated system under a simultaneous dual-fuzzing assault.

To assess the effectiveness of each fuzzer, a set of metrics was employ, with crash time serving as the cornerstone. Crash time precisely measures the duration from the initiation of fuzzing to the occurrence of a system crash. This metric evaluates a fuzzers effectiveness. Furthermore, utilized additional metrics, including:

- **Number of Crashes Encountered**: This metric quantifies the frequency of system crashes provoked by the fuzzer. Our experimentation involved 168 runs for each fuzzer and 54 runs for every ECU and bug type combination. This dataset provides insights into the fuzzer's capability.

In this case, a "crash" refers to the software being tested behaves unexpectedly or application shutdown. These crashes are essentially unintended consequences of fuzzing and are indicative of potential vulnerabilities or weaknesses within the system and may manifest as unexpected errors, system freezes, or even system shutdowns. Crash time was measured which quantifies how quickly a fuzzer identifies and triggers these unintended behaviors. A smaller crash time indicates a more effective fuzzer because it identifies vulnerabilities more rapidly.

- **Code Coverage Achieved During Fuzzing**: Measuring the depth of exploration within the system's codebase during testing, this metric showcases the fuzzer's capacity to identify potential bugs and vulnerabilities. Together, these metrics constitute a robust framework for evaluating each fuzzer's performance comprehensively. They enable us to conduct a thorough assessment of their prowess in the identification of vulnerabilities and the revelation of potential flaws within both the CAN and ROS systems.

Evaluation

Experimental Design and Data Collection

In our experimental design, the following three hypotheses was tested:

- **Hypothesis 1:** Fuzzing the unified system is more effective in uncovering vulnerabilities and inducing crashes than fuzzing its individual components in isolation.
- **Hypothesis 2:** There exists a statistically significant difference between the performance of the ROS fuzzer on the CAN bus compared to the CAN fuzzer in terms of crash induction and vulnerability detection.
- **Hypothesis 3:** There exists a statistically significant interaction between the fuzzing scenarios (CAN

Fuzzer, ROS Fuzzer, and CAN and ROS Fuzzer) and the types of bugs injected (dereferencing, buffer overflow, and divide by zero) concerning the ECU functions (signal, speedometer, door functions).

The experiments involved data collection using monitoring and logging tools, namely PCAN-View and rosbag. PCAN-View[17], a user-friendly software tool tailored for monitoring, analyzing, and logging CAN bus data, played an integral role in the real-time observation and analysis of raw CAN data throughout the CAN-to-ROS integration phase. Concurrently, Rosbag [18], a command-line utility within the ROS, was deployed for the recording, playback, and analysis of ROS message data disseminated across ROS topics during simulations.

To enhance the observational capabilities, a bash script was employed to monitor the simulation tool's behavior and record system crashes. The PCAN-View tool complemented this by providing additional data logging and validation, generating precise log files for each experimental case. Simultaneously, rosbag served as a supplementary data recording tool, ensuring redundancy and resilience in data collection. This approach to error detection enabled assessment of the performance and reliability of both the ROS fuzzer on the CAN bus and the CAN fuzzer. We analysed the data, including: crash times, system behavior patterns, and error logs.

This analysis was conducted to identify patterns or trends in the timing and nature of crashes, ultimately shedding light on the effectiveness of the fuzzing methodologies and their impact on the integrated system's robustness and reliability. Three statistical techniques were used: ANOVA (Analysis of Variance), Tukey-Kramer post hoc tests, and linear regression. ANOVA compares means among multiple groups, determining if significant differences exist between groups concerning a dependent variable. It is crucial in testing Hypothesis 1, which suggests that unified system fuzzing is more effective than individual components fuzzing in isolation.

Tukey-Kramer post-hoc tests identify specific groups with significant differences in means, helping to substantiate or refute Hypothesis 1. Linear regression is used to address Hypothesis 2, which asserts a significant difference between the performance of the ROS fuzzer on the CAN bus and the CAN fuzzer in terms of crash induction and vulnerability detection. This analytical approach helps quantify the extent of any performance disparity between the ROS fuzzer and the CAN fuzzer. The combination of these techniques provides a robust analytical framework for assessing hypotheses and gaining valuable insights into fuzzing scenarios' effectiveness in ensuring system robustness and security.

Results

The results now present the findings based on the three fuzzing scenarios (F1 (CAN Fuzzer), F2 (ROS Fuzzer) and F3 (CAN and ROS Fuzzer), comparing their effectiveness. The metric for effectiveness is the crash time, where a smaller crash time indicates a more effective fuzzer as it takes less time to find bugs. Statistical analysis, including ANOVA and Tukey-Kramer post hoc tests, provides robust support for Hypothesis 1. The ANOVA analysis reveals a significant impact of the Fuzzer factor (representing different fuzzing scenarios) on crash times ($p > 0.05$). This, in turn, underscores that there are statistically significant differences in crash times among the various fuzzing scenarios.

Delving deeper into these differences, our Tukey-Kramer post-hoc tests as seen in Figure 5 indicate several key findings. Firstly, a significant difference in crash times is observed between F1 (CAN Fuzzer) and F2 (ROS Fuzzer), implying distinct performances during the fuzzing process. Secondly, F2 (ROS Fuzzer) and F3 (CAN and ROS Fuzzer) exhibit a significant difference in crash times. However, intriguingly, no significant difference in crash times is found between F1 and F3. This outcome suggests that fuzzing both systems simultaneously (F3) doesn't significantly outperform fuzzing the CAN bus alone (F1) in terms of crash induction time.

Our linear regression analysis support for Hypothesis 2. The analysis reveals a significant F-statistic (51.84, p-value >2.2e-16), indicating a substantial influence of the choice of fuzzer on crash times. Specifically, Fuzzer 2 (ROS Fuzzer) demonstrates a positive association with "time," signifying longer crash times, and thereby suggesting a longer duration to uncover system vulnerabilities. In contrast, Fuzzer 3 (CAN and ROS Fuzzer) exhibits a negative association with 'Time', indicating its efficiency in swiftly identifying bugs within the system. Consequently, these results robustly support Hypothesis 2, emphasizing a statistically significant difference in crash times between the ROS Fuzzer (F2) and the CAN Fuzzer (F1).

In the evaluation of different fuzzers for testing the CAN-ROS system, Figure 6 depicted in the boxplots offer valuable insights into their respective performances. F3 stands out with a notably lower median and quartile values compared to F1 and F2. This implies that F3, on average, spent less time in identifying vulnerabilities within the system. The maximum values, excluding outliers, for F3 are also comparatively lower, indicating a

FIGURE 5 Multiple Comparisons of Means (Tukey Contrasts)

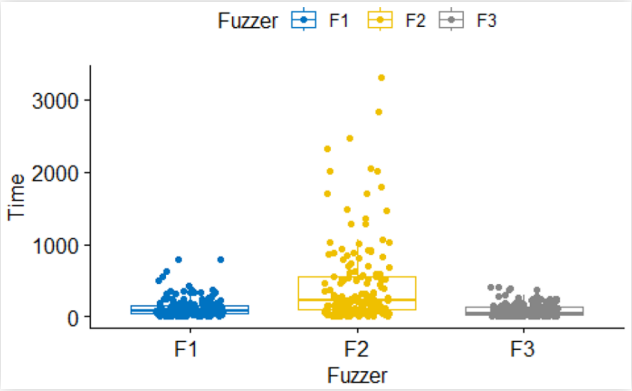
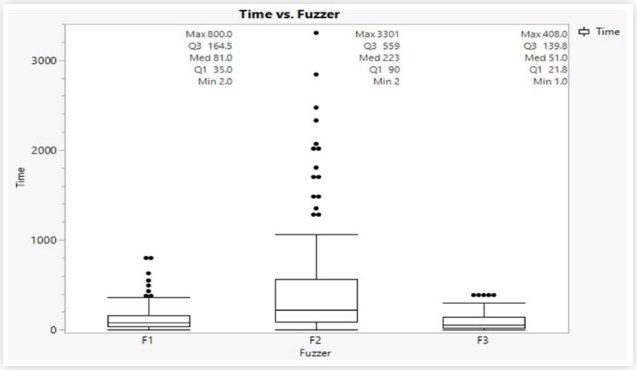


FIGURE 6 Boxplot for Fuzzers



consistent efficiency in the detection of bugs. In contrast, F1 and F2 exhibit higher median and quartile values, suggesting a relatively longer time to discover vulnerabilities. The boxplots collectively suggest that F3 is more effective in terms of time efficiency, potentially making it a preferable choice for future testing scenarios. These results illuminate the relative strengths of each fuzzer, providing a quantitative basis for understanding their performances in the context of the CAN-ROS system.

TABLE 1 Comparison Tables

F1: CAN Fuzzer			
Level	**Time**	**Mean**	**Std Error**
A	54	84.667	18.225
B	54	154.648	18.225
D	54	133.574	18.225
Door	8081	287.145	24.405
Signal	8899	324.921	23.256
Speed	3156	89.346	39.051
F2: ROS Fuzzer			
Level	Time	Mean	Std Error
A	54	324.537	77.863
B	54	513.944	77.863
D	54	489.685	77.863
Door	40882	1634.84	74.72
Signal	23256	699.01	99.07
Speed	7583	245.59	173.50
F3: CAN and ROS Fuzzer			
Level	Time	Mean	Std Error
A	54	78.574	12.388
B	54	108.241	12.388
D	54	76.426	12.388
Door	2930	202.046	17.560
Signal	5691	199.926	12.600
Speed	1467	84.175	24.817

Discussion

The shift towards increasingly automated and interconnected vehicle systems necessitates a proportional evolution in security testing methodologies. The diverse performance of the fuzzers in detecting different types of bugs reflects the complexity of modern vehicle systems, highlighting the need for comprehensive testing frameworks that can adapt to various components and their unique vulnerabilities.

The objective of this study was to evaluate and compare the performance of three different fuzzers — CAN Fuzzer, ROS Fuzzer, and ROS and CAN Fuzzer — in terms of their ability to discover bugs in our system. The primary criterion for assessing their effectiveness was the crash time, which reflects the time taken to uncover vulnerabilities or issues within the tested software.

As shown in Table 1, Fuzzer 1 was most effective at identifying vulnerabilities associated with the dereferencing bug, with buffer overflow exhibiting the longest mean crash time. This indicates that Fuzzer 1 is less effective at finding buffer overflow vulnerabilities. Signal had the longest mean crash time for Fuzzer 1, indicating its potential robustness in this context. The results indicate that Fuzzer 1 is capable of identifying vulnerabilities in a variety of test types.

The evaluation of Fuzzer 2 revealed distinct patterns in mean crash times for various categories of bugs and ECU function tests. According to the findings, the ROS Fuzzer had the longest crash times, which could be attributed to a number of factors, including the complexity of the ROS ecosystem, which may have more layers of abstraction than traditional CAN systems, potentially leading to a larger attack surface and more complex interactions that are more difficult to predict and test effectively. Also, the presence of the ROS to CAN translator node in the fuzzing process is notable. This node serves as a link between the high-level and complicated ROS messages and the low-level and protocol-constrained CAN frames. this node acts as a bridge between the ROS messages, which are typically high-level and complex, and the CAN frames, which are low-level and constrained by the CAN bus protocol. The translator node's role had implications for the effectiveness of the fuzzing process as the translation process introduces latency. Its longer crash times might be partly attributable to the complexity of translating ROS messages into CAN frames.

The findings suggest that the ROS Fuzzer had the longest crash times, which could be due to a variety of factors such as the complexity of the ROS ecosystem, which may have more layers of abstraction than traditional CAN systems, potentially leading to a larger attack surface and more complex interactions that are harder to predict and test effectively. A notable point is the presence of the ROS to CAN translator node into the fuzzing process, this node acts as a bridge between the ROS messages, which are typically high-level and complex, and the CAN frames, which are low-level and constrained by the CAN bus protocol. The translator node's role had implications for the effectiveness of the fuzzing process as the translation process introduces latency. Its longer crash times might be partly attributable to the complexity of translating ROS messages into CAN frames. The

fuzzing program must account for both the translation logic and the final CAN frames, which could explain why it took so long to find system flaws. This highlights the importance of comprehensive fuzzing methodologies that evaluate both the translation logic and the output.

The findings of the Fuzzer 3 investigation unveiled notable patterns in the average duration of crashes across several bug kinds and ECU function test types. The experimental results indicate that Fuzzer 3 had superior performance in discovering vulnerabilities, as evidenced by the lowest mean crash time seen between divide by zero and dereferencing pointer.

The fact that no significant difference in crash times was observed between the CAN Fuzzer and the combined CAN and ROS Fuzzer suggests that the CAN bus might be the limiting factor in these scenarios or that the combined approach does not synergistically improve the fuzzing process as might be expected.

Overall, Fuzzer 2 exhibited the highest mean crash times for all bugs, as well as in the ECUs. However, it's important to note that Fuzzer 3 outperformed Fuzzer 1. In terms of ECU function test types, Door consistently showed the highest mean crash times for both Fuzzer 2 and Fuzzer 3, while Signal demonstrated the highest mean crash time for Fuzzer 1, and in terms of bugs, buffer overflow was the longest, followed by divide by zero and dereferencing pointer.

Conclusion

In conclusion, the choice of a fuzzer plays a pivotal role in the efficiency of bug discovery within a system. The ability to achieve lower crash times is indicative of faster and more effective bug detection, a crucial factor in security testing and software vulnerability assessment. Based on the comprehensive findings of this study, Fuzzer 3, which simultaneously deploys both fuzzers, emerges as the most efficient and effective option among the three evaluated. This suggests that Fuzzer 3 is exceptionally well-suited for security testing tasks where rapid bug identification is of paramount importance.

While Fuzzer 1 and Fuzzer 3 did demonstrate competitive performance in certain scenarios, they generally exhibited shorter crash times, implying a comparatively swifter bug discovery process. It is important to note that the difference in performance between these two fuzzers and Fuzzer 3 is relatively slim, with an estimated gap of just 139 seconds. In contrast, Fuzzer 2 performed poorly across the board. Ultimately, the selection of the most suitable fuzzer should be driven by the specific requirements and objectives of the security testing project at hand. Fuzzer 3's superior performance, particularly in terms of faster bug discovery, positions it as a valuable tool for security professionals seeking efficient vulnerability assessment.

By addressing the specific challenges posed by collaborative driving, our future research is designed to not only enhance the efficiency and reliability of autonomous vehicular systems but also to rigorously examine the security aspects within these systems. A critical component of this future work will be an in-depth analysis of the CAN bus fuzzer's interaction with the ROS system. We explicitly aim to conduct an independent assessment of ROS, isolating its vulnerabilities. This focused exploration will be pivotal in identifying any inherent susceptibilities within ROS, thereby contributing significantly to fortifying the security of the integrated CAN-ROS environment. Our commitment is to provide a detailed and comprehensive understanding of the security implications at both the individual and integrated system levels, ensuring a robust and resilient framework for autonomous vehicular technologies.

References

1. Plot, J.A., "Red team in a Box (RTIB): Developing Automated Tools to Identify, Assess, and Expose Cybersecurity Vulnerabilities in Department of The Navy Systems," Ph.D. dissertation, Monterey, CA; Naval Postgraduate School, 2019.
2. Riebe, T., Schmid, S., and Reuter, C., "Meaningful Human Control of Lethal Autonomous Weapon Systems: The CCW-Debate and Its Implications for VSD," *IEEE Technology and Society Magazine* 39, no. 4 (2020): 36-51.
3. Sommer, F., Dürrwang, J., and Kriesten, R., "Survey and Classification of Automotive Security Attacks," *Information* 10, no. 4 (2019): 148.
4. Mehr, G., Ghorai, P., Zhang, C. et al., "X-Car: An Experimental Vehicle Platform for Connected Autonomy Research," *IEEE Intelligent Transportation Systems Magazine* 15, no. 2 (2023): 41-57, doi:10.1109/MITS.2022.3168801.
5. 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.
6. Manès, V.J., Han, H., Han, C. et al., "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering* 47, no. 11 (2021): 2312-2331, doi:10.1109/TSE.2019.2946563.
7. Elmadani, S., Nice, M., Bunting, M., Sprinkle, J. et al., "From Can to ROS: A Monitoring and Data Recording Bridge," in *Proceedings of the Workshop on Data-Driven and Intelligent Cyber-Physical Systems, ser. DI-CPS'21* (Nashville, TN: Association for Computing Machinery, 2021), 17-21, https://doi.org/10.1145/3459609.3460531.
8. Nice, M., Elmadani, S., Bhadani, R., Bunting, M. et al., "Can Coach: Vehicular Control through Human Cyber-Physical Systems," in *Proceedings of the ACM/IEEE 12th International Conference on Cyber-Physical Systems*, 2021, 132-142.
9. Haas, R.E., and Möller, D.P.F., "Automotive Connectivity, Cyber Attack Scenarios and Automotive Cyber Security," in *2017 IEEE International Conference on Electro*

Information Technology (EIT), 635-639, 2017, https://api.semanticscholar.org/CorpusID:7769642.

10. Oruganti, P.S., Appel, M., and Ahmed, Q., "Hardware-in-Loop Based Automotive Embedded Systems Cybersecurity Evaluation Testbed," in *Proceedings of the ACM Workshop on Automotive Cybersecurity, ser. AutoSec '19* (Richardson, TX: Association for Computing Machinery, 2019), 41-44, https://doi.org/10.1145/3309171.3309173.
11. Liu, R., Zheng, J., Luan, T.H. et al., "Ros-Based Collaborative Driving Framework in Autonomous Vehicular Networks," *IEEE Transactions on Vehicular Technology* 72, no. 6 (2023): 6987-6999, doi:10.1109/TVT.2023.3236978.
12. Koenig, N. and Howard, A., "Gazebo: Simulating Robots in a 3D Environment," *IEEE Robotics & Automation Magazine* 13, no. 3 (2006): 42-53.
13. PEAK-System Technik GmbH, *Peak pcan user manual* (PEAK-System Technik GmbH, 2022)
14. Linux CAN Utils, "Socketcan Userspace Utilities and Tools," accessed August 2023, https://github.com/linux-can/can-utils.
15. Caribou, C., "A Automotive Security Exploration Tool," accessed August 2023, https://github.com/CaringCaribou/caringcaribou.
16. SavvyCAN, "Can Bus Reverse Engineering and Capture Tool," accessed August 2023, https://www.savvycan.com/.
17. PEAK-System Technik GmbH, "PCAN-View," accessed August 2023, https://www.peak-system.com/PCAN-View.242.0.html.
18. Open Source Robotics Foundation, "ROS - Rosbag," 2022, http://wiki.ros.org/rosbag.

Contact Information

Iwinosa Aideyan,
Clemson University
iaideya@clemson.edu

Richard Brooks, Ph.D.
Clemson University
rrb@clemson.edu

Mert D. Pesé, Ph.D.
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

PUT - Program Under Test

CAN - Controller Area Network

ROS - Robot Operating System

ISSN 0148-7191