2026 IEEE/SICE International Symposium on System Integration (SII)
January 11-14, 2026. Cancún, México

Broken in Transit: Detecting Type Confusion in ROS 2 Deserialization via Fuzzing

Stephen Nwagwughiagwu[1], Jose Toribio[2], and Jeremy Blackstone[1]

Abstract— **The Robot Operating System 2 (ROS 2) has become the middleware backbone of modern robotics and cyber-physical systems, offering flexibility, modularity, and high-performance communication via the DDS protocol and eProsima's Fast-CDR serialization library. However, this reliance on implicit type contracts between publishers and subscribers introduces critical attack surfaces. In this paper, we present the first systematic study of type confusion vulnerabilities in ROS 2 deserialization, exposing a previously unexplored attack surface in robotic middleware. Through our fuzzing approach, we show that injecting malformed or mismatched message types into topics expecting a different format can trigger Fast-CDR deserialization failures. These failures propagate as uncaught exceptions resulting in process crashes and node-level outages.**

Our findings reveal a previously undocumented flaw in ROS 2's trust model for topic integrity, where the absence of runtime type enforcement or input validation leads to exploitable denial-of-service conditions. Through targeted fuzzing and case studies using standard ROS 2 messages, we evaluate the exploitability of this vulnerability in both simulation and physical robotics environments. This work underscores the need for secure-by-design messaging to ensure the reliability and safety of robotic middleware and cyber-physical systems.

I. Introduction

The rapid advancement of robotic systems has led to their increased adoption in various domains, including healthcare, transportation, and industrial automation. Robot Operating System (ROS) [1], [2], an open-source middleware for robotic applications, has emerged as a de facto standard, offering developers a powerful framework for designing, simulating, and deploying robotic systems. As one of the most powerful techniques in security analysis, fuzzing has played a pivotal role in uncovering high-impact vulnerabilities [3], [4]. It has been widely applied in several fields, including desktop software, network protocols, and Linux kernels. ROS 2 [5] nodes interact through middleware over networks, exchanging complex, typed data structures that are serialized and deserialized as they traverse the communication stack. While this design enables the rapid development of robust, flexible robotics applications, it also introduces new security and reliability concerns. In particular, deserialization type confusion, where received message bytes are incorrectly interpreted as the wrong type represents a dangerous class of potential vulnerabilities. Such type confusion may result in node crashes, unpredictable behavior, or even opportunities for remote code execution, posing a severe risk in safety-critical robotics deployments.

Despite efforts to improve ROS security, much of the existing research has focused on authentication mechanisms and network security [6], [7], [8], [9] , with relatively little emphasis on systematic methods for uncovering software vulnerabilities within ROS itself. Fuzzing is a dynamic analysis technique that involves automatically generating and injecting malformed or unexpected inputs into a system to uncover hidden bugs. Traditional fuzzing techniques like RVFuzzer [9] and PGFuzz [6] have been effective in identifying parameter errors and policy violations in drone controllers. Similarly, CPFuzz [10] has applied AFL-based coverage-guided greybox fuzzing to cyber-physical controllers. However, these approaches do not directly target type confusion vulnerabilities, which can lead to undefined behavior, memory corruption, or privilege escalation in ROS-based systems. This gap highlights the need for advanced testing techniques, such as fuzzing, to detect and analyze latent security flaws that may impact ROS-based robotic systems. In the context of ROS, fuzzing-based approaches can be instrumental in identifying type confusion vulnerabilities that may arise due to incorrect message handling, improper serialization/deserialization processes, or unsafe memory operations.

In this work, we demonstrate that deserialization type confusion and malformed message handling in ROS 2 can permit remote, input-triggered denial-of-service attacks on subscriber nodes. In contrast to traditional memory corruption exploits, attackers leveraging type confusion and improperly formatted DDS (Data Distribution Service) messages can reliably crash target ROS 2 nodes with a single, malformed input, often requiring no authentication or prior compromise. We show that these vulnerabilities are not merely theoretical: upstream DDS libraries such as Fast-CDR, which power the serialization layer of eProsima Fast-DDS [11] (the most commonly used middleware in ROS 2), demonstrably fail to handle malformed input, resulting in critical crashes when exploited in deployed systems.

Contributions: We make the following contributions:

- We uncover and exploit a previously undocumented ROS 2 subscriber crash vulnerability by crafting malformed DDS messages that subvert message type expectations.
- We show that the Fast-CDR serialization library, used

This work was supported by Amazon

[1]Stephen Nwagwughiagwu is a PhD Computer Science Student at Howard University, 2400 Sixth Street NW, Washington, DC 20059, USA stephen.nwagwughia@bison.howard.edu

[2]Jose Toribio is with the Department of Computer Science, Brown University, Providence, RI 02912, USA jose_toribio@brown.edu

[1]Jeremy Blackstone is an Assistant Professor with the Department of Electrical Engineering and Computer Science, Howard University, Washington, DC 20059, USA jeremy.m.blackstone@howard.edu

978-1-6654-5784-2/26/$31.00 ©2026 IEEE

pervasively in ROS 2 middleware, lacks resilience against malformed or inconsistent wire data, causing subscriber failures even in minimal nodes.

To our knowledge, this work presents the first comprehensive, practical demonstration of remote input-triggered crashes in mainstream ROS 2 deployments via deserialization bugs, and offers both immediate defenses and broader lessons for secure middleware design.

II. Related Work

This research is situated at the intersection of robotic middleware security, data serialization robustness, and the challenges of building resilient cyber-physical systems. Our work on type confusion vulnerabilities in ROS 2 builds upon, yet is distinct from, existing literature in these domains.

A. Security and Robustness in ROS2

The security of the Robot Operating System has been a topic of growing concern. Early work on ROS 1 established that the framework was fundamentally insecure by design, lacking authentication, authorization, or encryption, making it vulnerable to a wide range of network-based attacks [12].

With the advent of ROS 2, security was redesigned from the ground up by leveraging the Data Distribution Service (DDS) Security specification. This led to the development of SROS2 [8], which provides a comprehensive security framework for access control, encryption, and authentication. Research in this area has primarily focused on the correctness and implementation of these access control mechanisms. For instance, Deng et al. [13] performed a systematic security analysis of SROS2 and identified several critical design flaws related to policy synchronization and default misconfigurations that could allow an adversary to bypass access controls. While this line of work is critical for securing the system perimeter, it operates under the assumption that authenticated participants are well-behaved. Our research addresses an orthogonal threat: a legitimate but faulty participant that can destabilize the system not by violating access control, but by transmitting a structurally valid but semantically mismatched message.

Another significant area of research is the application of fuzz testing to ROS nodes and applications. Fuzzing frameworks like ROZZ [14], ROFER [15] , R2D2 [14], and RoboFuzz [16] have been developed to automatically discover bugs in ROS application code. These tools are powerful for finding memory errors, uncaught exceptions, and even semantic correctness bugs in the application logic of ROS nodes. However, they typically focus on the behavior of the application code itself. Our work differs by targeting a vulnerability within the middleware's deserialization layer, which occurs before the message data even reaches the application's callback function. This represents a more fundamental, pre-application vulnerability that traditional node-level fuzzers may not be designed to detect.

B. Data Serialization and DDS Middleware

Type mismatch and data serialization issues are a well-known class of vulnerabilities in distributed systems. Research outside the robotics domain has extensively documented how deserialization of untrusted data can lead to logical errors, denial of service, and even arbitrary code execution. The DDS standard, which underpins ROS 2, aims to mitigate these issues through a strictly-typed interface definition language (IDL). In theory, the IDL contract should prevent type confusion between publishers and subscribers. Studies on DDS middleware in the context of ROS 2 have largely focused on performance, latency, and throughput comparisons between different vendor implementations (e.g., Fast DDS, Cyclone DDS, Connext DDS) [17]. These analyses are essential for real-time system design but do not typically investigate the security implications of the middleware's implementation details. Our research demonstrates that while the DDS standard provides a specification for type safety, a brittle implementation of the serialization library (e.g., eProsima's Fast-CDR) can fail catastrophically when this specification is violated, even within a single DDS domain. Our work is the first, to our knowledge, to specifically demonstrate a denial-of-service vulnerability in ROS 2 stemming directly from a type mismatch at the DDS serialization layer.

C. Comparison to Broader Work in Integration Platforms and Cyber-Physical Systems

1) ROS 2 as an Integration Platform : ROS 2 is widely considered the de facto integration platform for modern robotics, enabling the composition of complex systems from heterogeneous software and hardware components. A key challenge in any integration platform is ensuring robustness at the interfaces between components. Our research highlights a critical failure at the most fundamental interface in ROS 2: the message-passing transport layer.

Our work complements broader CPS security research by introducing a practical example of the *"trusted but faulty"* threat model.

III. Motivation & Problem Statement

Robot Operating System 2 (ROS 2) has become the de facto standard for developing complex and distributed robotic systems, enabling the seamless integration of components from different vendors and research groups. In modern robotics from autonomous vehicles to collaborative manufacturing arms, a single system integrates numerous specialized subsystems: perception stacks, motion planners, safety monitors, and high-level mission coordinators. The success of such integration hinges on the reliability and robustness of ROS 2's underlying communication middleware, the Data Distribution Service (DDS), which is responsible for marshalling and delivering data between these distributed nodes.

The consequences of deserialization type confusion in ROS 2 are profound. In the best case, type confusion might cause a node to crash, disrupting robot functionality; in the

worst case, such flaws could open the door to data corruption, undefined behavior, or even remote compromise in safety-critical systems jeopardizing human safety and operational integrity [16], [14].

A. Overview of ROS2 Architecture

The Robot Operating System (ROS) is an open-source framework that enables developers to create modular and distributed robotic applications. It offers a comprehensive set of tools and libraries compatible with multiple operating systems and programming languages. Coupled with an active global community, ROS has built a strong ecosystem and gained widespread adoption [18], [19] becoming the de facto standard in robotics.

1) Data Distribution Service: The Data Distribution Service (DDS) is a well-established middleware protocol used in ROS 2 for real-time communication. It implements the Data-Centric Publish-Subscribe (DCPS) model [20]. In DCPS, a global data space stores all data objects known as DDS topics which function similarly to ROS topics and can be accessed by DDS processes. A process that publishes or subscribes to a topic is called a participant. Participant interactions are governed by configurable Quality of Service (QoS) parameters that define DDS behavior.

ROS 2 interfaces with DDS through abstract DDS APIs as shown in Figure 1. User-level code defines application logic, such as node communication patterns and message handling. This logic is processed by the ROS 2 Client Library (RCL) to create the node-based communication structure. The ROS 2 DDS Middleware (RMW) then translates this structure into the corresponding DDS configuration and parameters, which are passed to the DDS APIs to set up the DDS system.

This process establishes a one-to-one mapping between ROS 2 nodes and DDS participants. At runtime, when a ROS 2 node publishes a message, ROS 2 converts this action into DDS API calls, and the actual data transfer occurs within DDS. In this architecture, ROS 2 serves as a middleware layer and does not handle low-level protocol details.

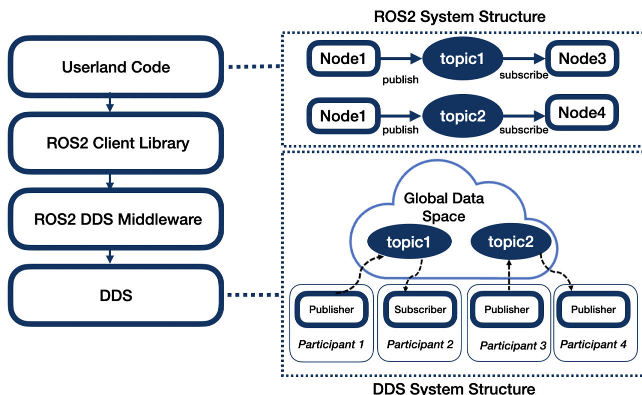

Fig. 1. ROS2 DDS architecture with the DCPS protocol

B. Fast DDS and Fast-CDR

- **Fast DDS:** EProsima Fast DDS library provides both an Application Programming Interface (API) and a communication protocol that deploy a Data-Centric Publisher-Subscriber (DCPS) model, with the purpose of establishing efficient and reliable information distribution among Real-Time Systems [1].
- **Fast-CDR:** EProsima Fast DDS uses a static low-level serialization library, Fast-CDR, a C++ library that serializes according to the standard CDR serialization mechanism defined in the Real-Time Publish Subscribe (RTPS) specification [2].

IV. Methodology

In our fuzzing technique, the node pretends to be a normal ROS 2 publisher on `/turtle1/cmd_vel` (type `geometry_msgs/Twist`) and alternates between:

- a valid `Twist` message, and
- a malformed payload created by serializing a `std_msgs/String` and then deserializing those bytes as if they were a `Twist` as illustrated in Figure 2.

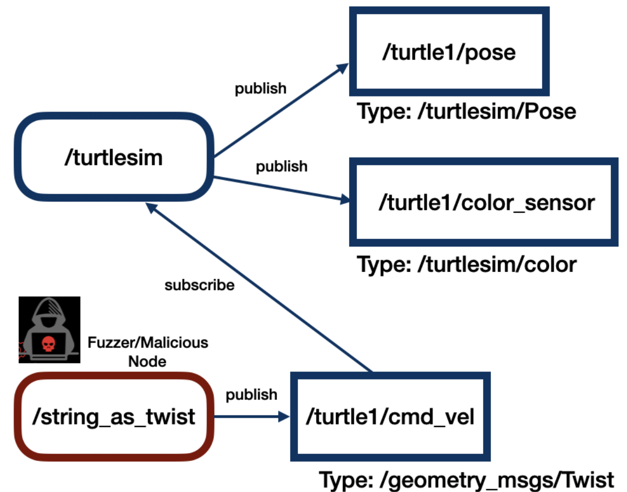

Fig. 2. Publish-subscribe model for the turtlesim with our fuzzing technique

We implement an intentional type confusion by trying to slip bytes that were never a `Twist` through the type boundary so downstream code or middleware may misinterpret them. In more detail, our fuzzing technique includes:

- It creates a publisher: `create_publisher(Twist, '/turtle1/cmd_vel'`, 10)
- Every 1.0 s it publishes a well-formed `Twist` with `linear.x=2.0` and `angular.z=1.0`.
- Every 1.5 s it builds a String message ("Hello N, disguised as Twist!"), serializes it with

[1]https://fast-dds.docs.eprosima.com/en/latest/02-formalia/titlepage.html
[2]https://www.omg.org/spec/DDSI-RTPS/

rclpy.serialization.serialize_message, and immediately deserializes those bytes using the Twist type.

In conclusion, our fuzzer manufactures a Twist object from bytes that came from a different message schema. That violates the intended contract of ROS 2 typing even though the publisher API still sees a `Twist` instance. Figure 3 shows the overall workflow for the exploit design.

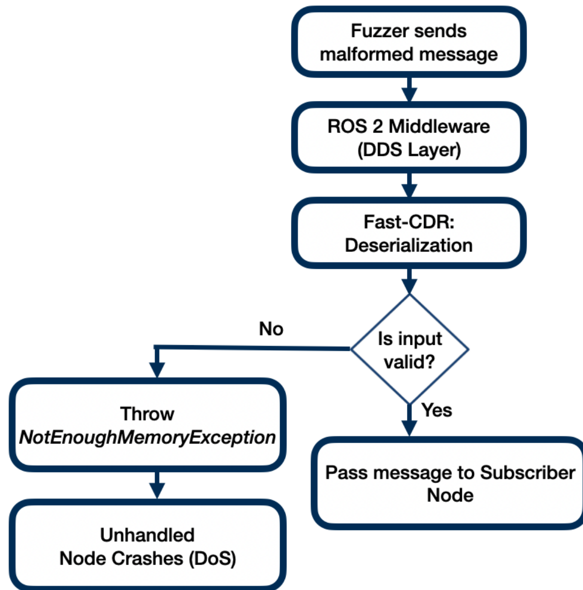

Fig. 3. Workflow showing how malformed messages propagate through ROS 2 middleware and Fast-CDR deserialization, potentially triggering unhandled exceptions that result in node crashes

V. Experiments

All experiments were conducted on an Alienware R5 laptop equipped with an AMD Ryzen9 6900HX mobile processor (8 cores, 16 threads, 3.3GHz base, 4.9GHz boost), an NVIDIA GeForce RTX3060 Mobile GPU, 32GB DDR5 RAM, and a 500 GB PCIe NVMe SSD. The base station hosted Docker containers running ROS2 Foxy, Humble and Jazzy providing a containerized environment for publishing and subscribing to topics with other ROS-enabled robots on the local network.

All experiments ran on physical hosts with Intel Xeon Gold 5218 CPUs and 256GB RAM under Ubuntu 22.04 LTS. ROS 2 Foxy [21], Jazzy and Humble with unmodified, default DDS middleware were used unless specified. All tests were repeated 5 times to reduce randomness.

A. Results

1) Test Targets: To comprehensively assess both the exploitability and real-world impact of ROS message type confusion vulnerabilities, we selected three representative robotic platforms spanning simulation, aerial, and ground robotics domains.

- *Turtlesim (simulation)* [22]. Turtlesim is a simple turtle simulator with a graphical window showing a turtle robot. The turtle can be moved using ROS commands or the keyboard.
- *DuckieTown PiDrone (aerial/embedded)* [23]. The Duckiedrone, often referred to as the PiDrone, is Duckietown's Raspberry Pi–based autonomous quadcopter platform. It is designed as an educational and research tool for hands-on learning of aerial robotics, control systems, and ROS-based autonomy.
- *Boston Dynamics Spot (commercial quadruped):* Spot is a four-legged, agile robot developed by Boston Dynamics, designed to operate both indoors and outdoors. It made its debut as the company's first commercially available robot in 2019, following a successful early-adopter leasing program.

Each platform was chosen to reflect a different use case, architecture, and level of operational complexity, and enabled controlled evaluation of malformed message attacks.

Turtlesim: As an initial testbed, we used the Turtlesim simulation environment included with ROS 2 Foxy. Turtlesim provides a canonical, deterministic benchmark for publisher-subscriber evaluation and message injection experiments as shown in Figure 4. We ran the default motion controller with `ros2 run turtlesim turtlesim_node`, and targeted the `'/turtle1/cmd_vel'` topic, which carries `geometry_msgs/msg/Twist`-typed velocity commands between publisher and subscriber nodes. As a result, there was an attempt to read or write more data than the buffer could hold during deserialization. Hence, causing the node process to crash as shown in Figure 5.

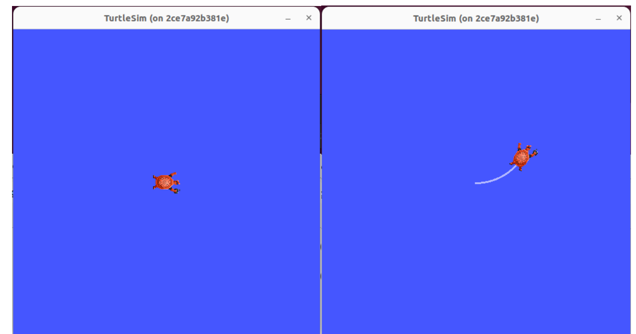

Fig. 4. Comparison of the Turtlesim initial state and its state after fuzzing. The turtlesim spawns *normally* but halts execution when a malformed message triggers a deserialization failure.

```
root@e8813e0dcd49:/# python3 turtle_confusion.py
[INFO] [1754611396.748359792] [string_as_twist_publisher]:
[VALID] Published real Twist #0
terminate called after throwing an instance of 'eprosima::f
astcdr::exception::NotEnoughMemoryException'
  what():  Not enough memory in the buffer stream
Aborted (core dumped)
```

Fig. 5. Runtime error thrown by Fast CDR. This exception indicates insufficient buffer memory in the stream, preventing successful deserialization.

DuckieTown PiDrone: To establish a physical robot baseline, we transitioned to the DuckieTown PiDrone, an ed-

ucational quadcopter running ROS 1 Kinetic on a Raspberry Pi 4 (4GB RAM). The PiDrone's modular IMU stack exposes the `/pidrone/imu` topic, which uses the standard `sensor_msgs/Imu` type to communicate inertial data (roll, pitch, yaw, accelerations, angular velocities) between the IMU node and downstream controllers. Using Rosbridge [24], we remotely published malformed IMU messages to `/pidrone/imu` in order to trigger type confusion and exceptions during flight. The drone initially responded correctly to a sequence of valid control messages, adjusting its orientation and maintaining stable flight. Once the malformed message was introduced, the control node crashed, causing the drone to halt abruptly as all propeller activity ceased.

Boston Dynamics Spot (ROS 2 Wrapper): For ground-based, high-complexity robotics validation, we interfaced with Boston Dynamics Spot using a ROS 2 wrapper that translates Spot's proprietary protocol, SDK, and services into standard ROS topics, actions, and services. Upon successful connection over Wi-Fi or payload link, and with control acquired, the wrapper exposes topics including:

- `/spot/joint_states` (publishes joint positions, velocities, efforts using `sensor_msgs/JointState`)
- `/spot/cmd_vel` (accepts `geometry_msgs/Twist` velocity commands)

We assessed the vulnerability surface by injecting malformed `sensor_msgs/JointState` messages into `/spot/joint_states` via the ROS wrapper to induce type confusion. This real-world evaluation [3] allowed us to observe how Spot's hardware reacted to corrupted joint-state inputs. Through this approach, we established a hardware-anchored baseline for identifying joint-level vulnerabilities. Figure 6 illustrates the PiDrone operating normally while receiving valid Twist commands, until it encounters a single malformed message at approximately 3.5 seconds, resulting in a system crash. This behavior indicates that the failure stems not from the command's value, but from the way the message is structured and processed internally.

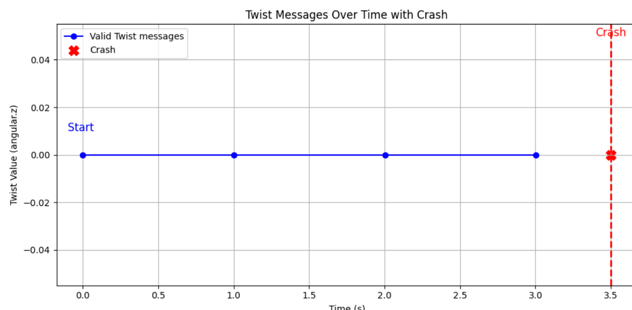

Fig. 6. Twist message sequence showing stable operation until a malformed message at 3.5 seconds triggers a crash despite unchanged angular values.

[3]PiDrone and Boston Dynamic Spot experiment video available at https://youtu.be/eJMaSm4N6Pg

VI. Discussion

Our research demonstrates that type confusion during message deserialization is not a theoretical edge case but a practical and potent vulnerability within the ROS 2 ecosystem. The ability to remotely crash a critical node with a single malformed message, authenticated or not, fundamentally undermines the principles of robust system integration. This discovery moves the conversation about ROS 2 security beyond high-level access control (SROS2) and into the low-level, high-impact domain of middleware resilience.

A. Broader Implications for System Integration

The implications of this vulnerability class are particularly severe in the industrial and safety-critical domains where ROS 2 is increasingly being adopted.

1) Industrial Automation and Manufacturing: In a modern factory, a robotic workcell is a tightly integrated system of manipulators, vision sensors, programmable logic controllers (PLCs), and safety systems. The failure of a single component can halt an entire production line, leading to significant financial losses. Our findings show that a buggy software update on a non-critical component, like a logging node, could inadvertently publish a message that crashes the primary manipulator controller. This transforms a minor software issue into a major physical and financial one. The "trusted but faulty" actor model is highly relevant here; the threat is not necessarily a malicious attacker but an internal, misconfigured component that can trigger a cascading failure across the integrated system.

2) Safety-Critical Systems (Autonomous Vehicles, Medical Robotics): In safety-critical applications, the consequences of an abrupt node failure are unacceptable. An autonomous vehicle's perception system, for instance, integrates data from LiDAR, cameras, and RADAR nodes. If a LiDAR driver node crashes due to a malformed message from a RADAR node, the vehicle's "world model" becomes incomplete, potentially leading to a catastrophic failure to detect an obstacle. The vulnerability we identified violates the core principle of freedom from interference, a key requirement for safety certifications like ISO 26262. A system cannot be considered safe if a non-critical component can cause the termination of a safety-critical one. Furthermore, this vulnerability provides a simple, effective vector for a malicious actor who has gained access to the internal network to cause physical harm by selectively disabling critical functions (e.g., braking or steering controllers) without needing to compromise the application logic itself.

VII. Conclusion & Future Work

In this paper, we introduced a deserialization exploit methodology targeting the ROS 2 message transport layer. Our framework successfully uncovered multiple previously unknown bugs, highlighting the systemic risks posed by unchecked type confusion and deserialization flaws in robotic middleware. These findings underscore the potential for real-world safety hazards and service interruptions when message integrity is compromised. While our findings are significant,

it is important to acknowledge the limitations of this study and outline avenues for future research.

- *DDS-Vendor Specificity:* Our analysis primarily focused on eProsima Fast DDS, the default middleware for many ROS 2 distributions. The specific `NotEnoughMemoryException` is an artifact of its Fast-CDR serialization library. While the vulnerability class of type confusion is middleware-agnostic, the manifestation of the bug may differ across other DDS implementations like RTI Connext or Eclipse CycloneDDS. It is plausible that other vendors' implementations might fail in different, potentially more insidious ways such as silent data corruption instead of a loud crash. A comparative fuzzing analysis across all major DDS vendors is a critical next step to create a comprehensive threat model for the ROS 2 ecosystem.
- *Beyond Denial of Service (Unhandled Edge Cases):* Our research successfully demonstrated a reliable method for causing Denial of Service. However, other, more severe consequences of type confusion remain unexplored.

Although arbitrary code execution was not achieved, crafted payloads exploiting heap overflows in deserialization may enable it. Exploring this risk via DDS messages remains a critical direction for future work. We reported the identified type confusion vulnerability to the middleware vendor. The disclosure was made under responsible disclosure practices following their published security policies. At the time of submission, the maintainers acknowledged the issue and are investigating potential mitigations.

In conclusion, for ROS 2 to fulfill its promise as a trustworthy foundation for the next generation of integrated robotic systems, its foundational communication layer must be hardened. Our work serves as a call to action for the ROS 2 community and DDS vendors to move beyond simple crash fixes and invest in comprehensive, security-first engineering practices, including robust input validation, graceful error handling, and continuous fuzz testing at the middleware layer. Access our fuzzer on GitHub: *https://github.com/SecurityReHU/FuzzingROSTypeConfusion*.

Acknowledgement

This work was supported primarily by an Amazon gift award. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of Amazon.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, 2009, p. 5.

[2] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[3] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, pp. 118–137, 2018.

[4] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[5] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074

[6] H. Kim, M. O. Ozmen, A. Bianchi, Z. B. Celik, and D. Xu, "Pgfuzz: Policy-guided fuzzing for robotic vehicles." in *NDSS*, 2021.

[7] B. Breiling, B. Dieber, and P. Schartner, "Secure communication for the robot operating system," in *2017 annual IEEE international systems conference (SysCon)*. IEEE, 2017, pp. 1–6.

[8] R. White, D. H. I. Christensen, and D. M. Quigley, "Sros: Securing ros over the wire, in the graph, and through the kernel," *arXiv preprint arXiv:1611.07060*, 2016.

[9] T. Kim, C. H. Kim, J. Rhee, F. Fei, Z. Tu, G. Walkup, X. Zhang, X. Deng, and D. Xu, "{RVFuzzer}: Finding input validation bugs in robotic vehicles through {Control-Guided} testing," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 425–442.

[10] F. Shang, B. Wang, T. Li, J. Tian, and K. Cao, "Cpfuzz: Combining fuzzing and falsification of cyber-physical systems," *IEEE Access*, vol. 8, pp. 166 951–166 962, 2020.

[11] G. Sciangula, D. Casini, A. Biondi, C. Scordino, M. Di Natale *et al.*, "Bounding the data-delivery latency of dds messages in real-time applications," *LEIBNIZ INTERNATIONAL PROCEEDINGS IN INFORMATICS*, vol. 262, 2023.

[12] B. Dieber, B. Breiling, S. Taurer, S. Kacianka, S. Rass, and P. Schartner, "Security for the robot operating system," *Robotics and Autonomous Systems*, vol. 98, pp. 192–203, 2017.

[13] G. Deng, G. Xu, Y. Zhou, T. Zhang, and Y. Liu, "On the (in) security of secure ros2," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 739–753.

[14] Y. Shen, J. Liu, Y. Xu, H. Sun, M. Wang, N. Guan, H. Shi, and Y. Jiang, "Enhancing ros system fuzzing through callback tracing," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 76–87.

[15] J.-J. Bai, H.-X. Song, and S.-M. Hu, "Multi-dimensional and message-guided fuzzing for robotic programs in robot operating system," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 763–778.

[16] S. Kim and T. Kim, "Robofuzz: fuzzing robotic systems over robot operating system (ros) for finding correctness bugs," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 447–458.

[17] T. Kronauer, J. Pohlmann, M. Matthé, T. Smejkal, and G. Fettweis, "Latency analysis of ros2 multi-node systems," in *2021 IEEE international conference on multisensor fusion and integration for intelligent systems (MFI)*. IEEE, 2021, pp. 1–7.

[18] "ROS Community Metrics Report," http://download.ros.org/downloads/metrics/metrics-report-2020-07.pdf, 2020, accessed: 2025-08-14.

[19] "Ros robots," https://robots.ros.org/, 2022.

[20] G. Pardo-Castellote, B. Farabaugh, and R. Warren, "An introduction to dds and data-centric communications," *Real-Time Innovations*, vol. 61, 2005.

[21] Open Source Robotics Foundation, "ROS 2 Foxy Fitzroy," https://docs.ros.org/en/foxy/Releases/Release-Foxy-Fitzroy.html, 2020, accessed: July 25, 2025.

[22] École Nationale Supérieure des Mines de Saint-Étienne, "TurtleSim Tutorial," https://www.emse.fr/ boissier/enseignement/defiia/up9-23/turtlesim.html, 2023, accessed: July 31, 2025.

[23] Duckietown, Inc., "The Duckiedrone (DD24) Operation Manual," https://docs.duckietown.com/daffy/opmanual-dd24/intro.html, 2025, accessed: July 31, 2025.

[24] C. Crick, G. Jay, S. Osentoski, and O. C. Jenkins, "Ros and rosbridge: Roboticists out of the loop," in *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction*, 2012, pp. 493–494.