Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Mário Macedo Ribeiro

Formalizing ROS2 security configuration with Alloy

December 2021

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Mário Macedo Ribeiro

Formalizing ROS2 security configuration with Alloy

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
Manuel Alcino Pereira da Cunha
André Filipe Faria dos Santos

December 2021

ACKNOWLEDGEMENTS

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

Automation developments are becoming essential to industrial restructuring, as it brings more efficient and accurate processes with less associated cost. Consequently, robots are rapidly being deployed in a wide range of scenarios, especially where security is demanded. In such cases, it is critical to employ appropriate procedures to verify both the system's quality and its security.

Following the current growth of cyber-physical system, as well as their usage into various technology domains, the development of software applications is demanding due to the complexity behind the integration of needed services, beyond those provided by the operating system. Hereupon, middleware software to abstract systems hardware are constantly evolving, while offering services that support application development and delivery.

One of the most popular open-source software platforms for building robotic systems is the Robot Operating System (ROS) [35] middleware, where highly configurable robots are usually built by composing third-party modules. A major factor behind its popularity and widespread adoption is its flexibility and interoperability. One drawback of this flexibility, however, lies in the increased security risks that ROS applications face. Alongisde security risks, the arising of performance and scalability issues related to the ROS middleware specification, forced the creation of a new version of ROS.

Robot Operating System 2 (ROS2), which continues to provide a simple, uniform message passing interface to allow components to communicate with each other, now implemented using the Data Distribution Service (DDS) [30] communication protocol, where security guarantees are ensured by DDS-Security specification. Using DDS-Security, it is possible to configure ROS2 to run with security guarantees using the SROS2 toolset [17]. However, improper configuration can still lead security problems.

This thesis proposes a technique, based on the software verification perspective, to automatically verify system-wide properties related to the security configuration of ROS2-based applications. The intended purpose is to model the ROS architecture, as well as the network communication behaviour, in Alloy [19], a formal specification language and analysis tool supported by a model-finder over which, system-wide properties are subsequently model-checked.

KEYWORDS Robotics, ROS, ROS2, DDS, SROS2, Security, Software Verification, Alloy

RESUMO

A constante implementação da ideia de automização de processos tem motivado a reestruturação nos mais diversos setores industriais, com o obejtivo de aumentar a eficiência e precisão nos processos integrados, consequentemente, reduzindo os custos associados. Consequentemente, impulsionam a integração robótica nos mais amplos domínios tecnológicos, especialmente em domínios onde a segurança é exigida. Nestes cassos, é fundamental adotar técnicas apropriadas de forma a verificar tanto a qualidade do sistema, como a segurança do mesmo.

Como resultado do atual crescimento de sistemas sistemas ciber-físicos, nomeadamente sistemas robóticos, bem como sua utilização em vários domínios tecnológicos, o desenvolvimento de aplicações é exigente devido à complexidade da integração dos serviços necessários, tipicamente não fornecidos pelo sistema operativo. De forma a acompanhar o aumento na complexidade destes sistemas, middlewares que permitem abstrair hardware têm sido adoptados, oferecendo serviços que oferecem suporte ao desenvolvimento de aplicações robóticas.

Uma das plataformas considerada como *standard* no que toca ao desenvolvimento sistemas robóticos é o middleware Robot Operating System (ROS) [35], onde robôs altamente configuráveis são construídos atráves da composição modular de *software* externo, oferencedo características como flexibilidade e interoperabilidade aos sistemas integrados. No entanto, a flexibilidade resulta num aumento de vulnerabilidades de segurança, pondo em causa a integridade das aplicações. Além da falta de segurança apresentada, existem também problemas de desempenho e escalabilidade relacionados com a especificação do middleware. Assim, era necessário uma mudança na estruturação do ROS, resultando na criação do Robot Operating System 2 (ROS2).

A nova versão do ROS, o Robot Operating System 2 (ROS2) implementa um protocolo de comunicação de nome Data Distribution Service (DDS) [30], onde para além de garantir serviços de comunicação, fornece diversas especificações, sendo uma delas a especificação DDS-Security, que através de uma metodologia de *plugins*, oferece diferentes métodos de adoção de segurança. Através do uso desta especificação, juntamente com o uso do SROS2 *toolset* [17], é possível configurar o ROS2 de forma a adotar estas medidas de segurança.

Esta tese propõe uma técnica para a verificação automática de *system-wide properties* em aplicações ROS. Esta técnica apresentada baseia-se na formalização estrutural de arquiteturas ROS em Alloy [19], com o obejtivo de modelar o comportamento associado à comunicação dentro do sistema, tendo em consideração configurações associadas às propriedades de segurança.

PALAVRAS-CHAVE Robótica, ROS, ROS2, DDS, SROS2, Segurança, Verificação de Software, Alloy

CONTENTS

LIST OF FIGURES

Part I

INTRODUCTORY MATERIAL

1

INTRODUCTION

The concept behind automation development is being incorporated into the industrial world, through the use of flexible tools to assist in the most various scnearios, as it brings efficiency and accuracy to the industry's processes. Robots have emerged into our world as they are used routinely to perform quests done traditionally by human beings. The decreasing cost of robots is one of the main reasons for their increased use. [7] Economics aside, the usage of robots brings increased productivity, safety and more manufacturing production work back to developed countries. [16] Robotics is already the key driver of competitiveness and flexibility in large scale manufacturing industries, as it is significantly reliant on a variety of technologies, connected into the same robotic system. Due to the continuous growth of technology in these different domains, robots can be used in a wide range of applications. [29]

Despite the advances in technology, dealing with hardware-level applications becomes highly impractical as the complexity systems increases. Thereupon, developing and writing software code for robot applications is demanding, where multiple aspects must be considered properly [34], as robots became to be integrated into distributed systems through seperated components distributed through the network, connecting different hardware and software modules raises interoperability and communication issues. To solve this issue, modular architectures based on message-passing communication patterns are continually emerging as a middleware layer, offering services to the application layer, easing the development cost, while providing interoperability and communication facilitation. [29, 27]. The requirement for a middleware layer that meets different robot's specfication is a novel approach to enable the creation of robot applications over robotic systems, while supporting features such as robustness and modularity.

The Robot Operating System was created by a collaborative open-source community to contribute in the advancement of robots [13] that enables locomotion, manipulation, navigation, and recognition tasks, complemented by software libraries and tools with the aim of helping build robot applications easily, without the need of building it from scratch. ROS was designed to be flexible, since working with robots requires experimentation, given the variety of different hardware and software, enabling interpolation with potential added components. However, performance and scalability issues arised due to its middleware specification. [34] Additionally, real-time constraints such as fault-tolerance, deadlines, or process synchronization were not supported by ROS, making it unsuitable for safety-critical and real-time systems. [23]

Besides having no middleware support for distributed real-time systems, security features were also not featured by ROS, which started to be demanding for deployed systems. An increasing number of real-time applications, for

instance robotic systems, requires security ensurance for protecting real-time sensive data, against unauthorized access. [24]

This lead to the creation of Robot Operating System 2, which continues to provide a simple, uniform message passing interface to allow components to communicate with each other, now implemented using the Data Distribution Service (DDS) [30] specification protocol as its middleware, leveraging for its messaging architecture. Issues concerning system integration and scalability are mitigated by DDS various implementations, due to the several transport configurations provided, making it suitable for real-time distributed systems. DDS also provides a security specification, called DDS-Security, used by multiple implementations, and ROS2 makes use of them to provide features to the deployed robotic systems. [2]

With the extended use of robotic systems, software verification, through the use of formal methods, are necessary to prevent potentially catastrophic consequences, mainly related to security matters, as safety guards are gradually implemented into the software domain. Within this context, Alloy [20, 25] framework enables the expressness the behaviour of systems with rich configurations, due to the combination of both relational and linear temporal logic (LTL) provided by its specification language, consequently supporting model-checking techniques. Model-checking techniques enable far better levels of coverage and, as a result, more reliability than traditional testing, where the system is abstracted as a conventional model, that is automatically checked over performing property verification on finite-state machines. [3]

The proposal of this dissertation is to develop a novel technique to automatically verify system-wide safety properties using Alloy framework, confining a ROS2 system into an abstract model, in order to obtain a prototype tool that can be used by developers to easily detect security configuration issues on their respective robotic application.

1.1 OBJECTIVES AND CONTRIBUTIONS

The first goal of this thesis rests in introducing concepts around the Robot Operating System, contextualizing the evolution behind its framework towards achieving security, where the former version of ROS lacked due to the focus on flexibility. Since ROS2 has been developed over the DDS framework, as its communication middleware, DDS must be properly understood before considering the security aspects. To do so, simple examples will first be studied and introduced, in order to provide ROS-based context to the reader. The domain of autonomous systems, where security is of extreme relevance, namely the Autoware [22] ROS2 platform for self-driving vehicles will also be introduced. To understand SROS2 we intend to configure and run a realistic case study related to Autoware with security guarantees.

The DDS Security standard functionality is evaluated, as well as how security is integrated into ROS2. Since security issues, concerning public networks, are recent to the robotics domain, ROS2 security network design should be analysed structurally.

Security configuration related to SROS2 toolset will be provided in this chapter, supported by an example that accounts multiple security features, those being authentication, encryption and, most importantly, access control, applying restriction constraints to the network and its participants, that by default are not controlled.

The second goal is to extend a previously proposed [4] formalization of ROS applications in Alloy/Electrum [19, 25] to also take into consideration the security configuration defined with SROS2. Using this extension, we intend to explore the viability of verifying simple information-flow security properties. For instance, to ensure that no commands to the vehicle motor can be sent via the infotainment system.

The final goal is to automate the extraction of such formal Alloy models from the configuration files of a ROS2 application, in order to obtain a prototype tool that can be used by roboticists to easily detect security configuration issues.

1.2 DOCUMENT STRUCTURE

The current dissertation structure is divided into four different chapters. Chapter (2) introduces all the concepts related to Robot Operating System, and its evolution as robotic development framework towards achieving system security. The current provided work regarding ROS and security in ROS is also presented in this chapter. Chapter (3) introduces the Alloy framework, as its specification language supported by a concrete example case. Chapter (4) presents the approach developed during this work, which allows the automatic verification of system-wide safety properties in ROS2 applications. Chapter (5) presents the thesis conclusions, alongisde possible future aproaches complementing the current work.

2

ROBOT OPERATING SYSTEM

As aforementioned, this thesis aims to explore the security configuration within Robot Operating System 2 (ROS2) and its security toolset Secure Robot Operating System 2 (SROS2), enabling the specification and model checking using Alloy.

Robotic systems have emerged into several scenarios, where its usage ranges between basic processes automation, up to full performance over critical tasks, consequently causing the complexity increase in these domains. Concerning the complexity behind writting software code, due to the widely variety presented in the robot's hardware as in fields of action [6], Robotic Operation System (ROS) presents itself as a middleware system, created to facilitate robotic systems development.

In ROS, software flexibility was valued above all else, meaning that values like security were not considered as a priority, so ROS-based applications tend to face increased security risks, compromising the whole robotic network. As ROS became a standard for many robotic systems, and due to the scale and scope of the robotics grows as they start to be integrated into the real world, security matters must be addressed as priority while developing systems involving robots. [13, 23]

The upgraded version of ROS, Robot Operating System 2 (ROS), presents itself as a framework for developing robotic systems, supported by a standard, the Data Distribution Service (DDS), where multiple middleware implementations are built over this standard, providing ROS applications multiple DDS-based specfications, as well as valued Quality of Service (QoS) settings over the transport configuration.

The DDS-Security specification [31], embedded by every DDS implementation supported by ROS, supplies multiple plugins regarding the security domain. As result, ROS2 yields a wider command toolset compared to the former version of ROS, as they bring forth to a toolset, the Secure Robot Operating System 2 (SROS2) toolset, concerning the security functionality that DDS-Security plugins offer.

This chapter introduces necessary background information over the major concepts on which this thesis rests. First, it is presented a detailed introduction to the concepts around Robot Operating System (ROS), as well as the evolution approach that ROS faced towards providing security to its deployed systems. Regarding this goal, Data Distribution Service (DDS) and its integration on Robot Operating System 2 (ROS2) must be contextualized beforehand. Thereafter, it will be presented a section dedicated to related work regarding the study of security in ROS based systems.

2.1 FORMER ARCHITECTURE

The Robot Operating System was created by a collaborative open-source community, that has undergone rapid development [6], to contribute in the advancement of cyber physical systems, mainly robotic systems, serving as developer enhancer for the world of robotic applications [13], complemented by software libraries and tools with the aim of helping build robot applications easily, without the need of building it from scratch.

Although Robot Operating System furnishes services, often perceived as operating system's services, such as hardware abstraction, low-level device control and control over message-passing between processes, ROS can not be perceived as a proper operating system, in the sense of process management and scheduling. However, it has a significant impact on the deployed application's performance, with highly complex effects on timing leading to affecting the application's runtime behaviour. In result, the impact of the underlying operating system scheduling is over exceed by ROS. [34, 5]

Fundamentally, ROS is a middleware, as it provides a custom serialization format, a custom transport protocol as well as a custom central discovery mechanism, presenting itself as a distributed layer between the top application layer, the operating systems layer and its communication capabilities.

ROS was designed to provide as much as modularity and composability to the application layer [5] as possible, allowing ROS applications to be built over several software modules, as independent computing processes called *nodes*, that compose together to fulfill the deployment characteristics of the corresponding robot. [27]

The communication approach that ROS implements to perform data exchange between nodes is based on a publish-subscribe model, based on TCPROS [14] and UDPROS [15], that makes use of TCP and UDP sockets, respectively. [27] The information exchange is done by introducing the *message* definition, which characterizes every data structure concerning the information exchange between ROS participants. The publish-subscribe pattern brings various forms of decoupling and transparency. However, network security may be jeopardized, as a result of the transparency within publishers and subscribers, as the exchanged data can not be properly supervised. [23]

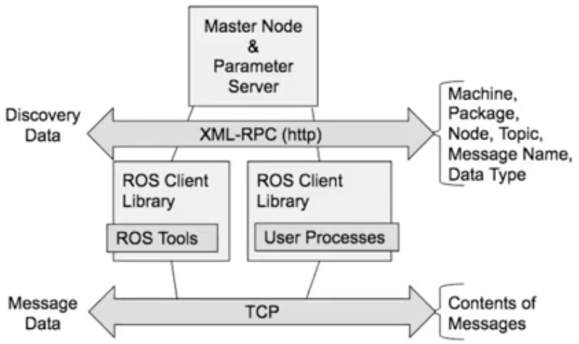

Figure 1: Robot Operating System architecture.

The former communication architecture supported by ROS focused on a centralization perspective, as it had a implementation of a *Master node*, that controlled every aspect of the communication establishment, meaning that

every information exchange between nodes had to go through this master. A ROS Master node must be initialized before any node, as these nodes must also be able to address the ROS Master's location. The ROS Master was also in charge of the parameters, which offer important runtime information about the network nodes.

Due to the sheer wide capabilities controlled by the master, this centralization approach fits the purposes of a research tool, as it is simpler to monitor and analyse the system behaviour. This communication architecture, however, does not scale well since it is heavily reliant on the master node's availability, making it unsuitable for safety-critical and real-time applications. If the master fails, the entire system fails, representing a single point of failure and a huge performance bottleneck.

Many research communities tried to fix these real-time issues by proposing potential solutions, while supporting the same architecture design, that did not fully accomplished the needs of real-time applications. So, it became clear to the ROS community that the framework had architectural limitations that could not be rearranged using the same design approach. [27]

Robot Operating System 2 comes as a complete refactoring of ROS, with the aim of increase the framework's real-time capabilities, by allowing the development of time-critical control over ROS, as it moves away from the former architectural design towards the implementation of an external middleware that can support the production needs of the outgrowing robotic systems. [23, 5]

2.2 EVOLUTION TOWARDS ROBOT OPERATING SYSTEM 2

The Robot Operating System 2 (ROS2) was developed with the goal of easing the former ROS architecture. Although ROS2 continues to provide a simple, uniform message passing interface to allow components to communicate with each other, instead of implementing their own middleware specification, Data Distribution Service (DDS) communication protocol is implemented as an abstract middleware interface, through which serialization, transport and discovery is being provided.

2.2.1 *Data Distribution Service*

The Data Distributed System [30], simply known as DDS, is a Object Management Group (OMG) middleware standard, resulted from the need of better interoperability between different vendors middleware frameworks, directly addressing data communication between nodes that belong to a *publish-subscribe* communication architecture, for real-time and embedded systems, that relies on a data-centric approach. DDS disposes a software API supported by a rich documentation about its exact behaviour. By furnishing this specification, it enables third parties applications, such as ROS, to implement this middleware, while auditing and understanding matters are covered by its documentation.

A middleware, such as DDS, is a software layer that lies between the operating system and the Application layer. [18] DDS is usually implemented by sytems that highly rely on timing-measures and efficiency, when it comes to data-delivering between nodes. DDS aims to ease the complexity behind creating each sytem's own middleware, by handling relevant aspects like network configuration, communication establishment, data sharing

and low-level details. As a result, system developers can mainly focus on their applications purposes, rather than concerning about information moving across levels.

DDS uses the Data-Centric Publish Subscribe (DCPS) model as its communication model approach. DCPS is based on a *publish-subscribe pattern*, where the *data-centric* messaging technique is implemented. It conceptually creates a virtual *global data space*, acessible by any DDS-based application. These applications, identified as *domain participants*, communicate with one another by data publishing and subscribing. [27]

In a data-centric architecture, the infrastructure formally defines the data and imposes rules over it, with the continuous awareness of the contents in the data space, as data is properly delivered to the applications which quest for it, saving bandwith and processing power. [30] Rather than forcing each application to deal with this complexity of defining the data space, DDS directly implements and provides controlled, managed, secure data sharing. [32]

A domain participant enables an application to participate in the global data space, either as a *publisher* or as a *subscriber*, according to their role on data transmission. A publisher is composed by *Data Writers*, where each of them publishes data to a corresponding *topic*. A *topic* represents the data space objects through which data is handled. A *Data Reader*, complementing the role of the data writer in the DCPS data transportation, declares the intent to subscribe to the topic. A subscriber can be composed by one or more data readers. [1, 9]

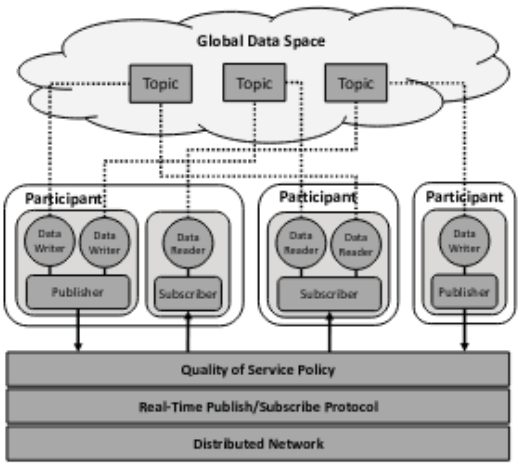

Figure 2: DDS architecture: DCPS model with RTPS. Extracted from [27].

To properly adress the data transportation through physical network, DDS offers a wire specfication protocol called Real-Time Publish-Subscribe Wire Protocol (RTPS) [37]. This protocol also works under a *publish-subscribe* policy over best-effort transports, where data transmission between endpoints is handled. [41] RTPS allows multiple applications, that could differ on their used DDS implementations, to interoperate with each other as network domain participants, by abstracting and optimizing transport. [9, 1] Besides, with the aim of easing the communication establishment, RTPS provides a set of discovery services, allowing the automatic discovery between participants. [9]

Furthermore, RTPS was designed to make use of *Quality of Service* profiles, where multiple transport policies can be specificed that, by default, DDS does not support. For instance, as multiple DDS vendors are built over the

UDP [33] transport protocol, which does not feature reliable delivery of data, transport reliability, often required by real-time environments, can be ensured by specifying the realiability's corresponding QoS policy into the RTPS communication layer. Besides, QoS profiles also enable security deployment into the transport configuration. This approach offers flexibility over communication configuration and development versatility, allowing the developer to specify whatever QoS satisfies its system communication needs. [13, 27]

Additionally, each domain entity manages the data according to the Quality of Service profiles speficified over the RTPS protocol, that is used over the transportation process as a policy. Applications within the network are then generated as DDS domain participants, with regard of the transport QoS policies. [1, 27]

Briefly speaking, DDS leverages the premise of a transport-independent virtualized Data Bus, to address the distribution of network resources, in which stateful data is distributed through the network. The involved applications can acess this data in motion, representing an architecture with no single point of failure, respectively enabling a realiable way of ensuring data integratity. Consequently, by adopting this approach, the load on the network is independent of the number of applications, making it easily scalable.

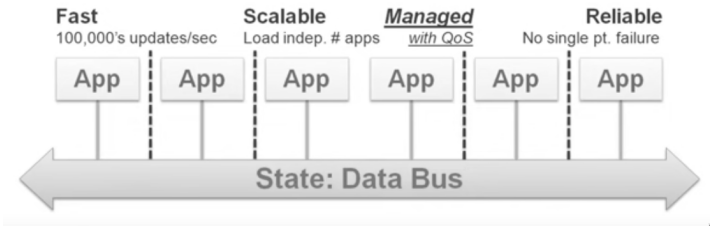

Figure 3: Data Distributed System architecture in a nutshell.

2.2.2 *ROS-DDS Architecture*

As previously stated, the Robot Operating System 2 was developed to address the lack of support for real-time systems that the former ROS provided, mainly due to its architecture design that relied on their own middleware specification. To address this, ROS2 middleware approach is built upon the DDS framework [27], leveraging DDS for its messaging architecture, where communication and transport configuration are handled. As a result of the DDS integration, ROS2 applications are actually considered DDS applications, meaning that is possible to interoperate both native DDS applications and ROS applications, providing flexible compatibility.

As far as dependencies are concerned, DDS implementations have light sized dependencies, often related to language implementation libraries, easing the complexity behind installing and running dependencies for ROS developers. [40]

The middleware's on-top layer regards the ROS client library (*rcl*), already implemented in the former ROS architecture. This layer regards the availability of ROS concepts to the Application layer, as it provides APIs to ease the software implementation by ROS developers. [36] As ROS aims to support different programming languages over the same computing context, each language-specficic API must have its corresponding client library (*rclcpp* regarding *C++* and *rclpy* regarding *Python*). The *rcl* accounts these client libraries by abstracting their specification, reducing code duplication. [38, 5]

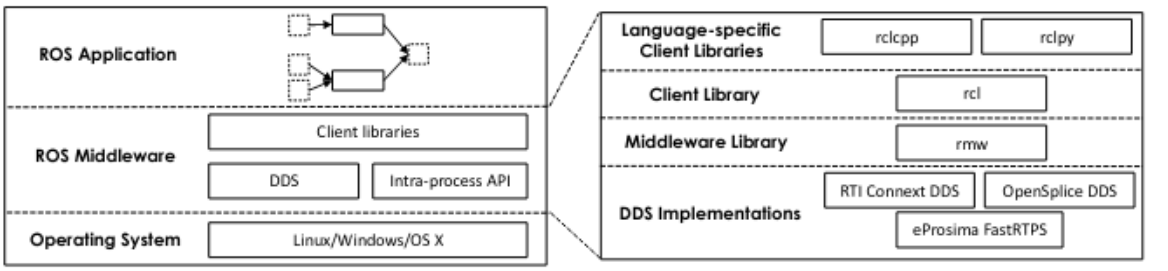

Figure 4: ROS2 framework architecture.

Towards supplying a wide range of configurations back to application layer, to vastly cover the robotic applications needs, ROS2 aims to support multiple DDS implementations, in which these implementation's API specification might differ from each other (currently, FastRTPS by eProsima, Connext by RTI, and Vortex OpenSplice by Adlink).

In order to abstract *rcl* from the specification's complexity of these implementation's APIs, an DDS-agnostic interface is being introduced, the rmw (ROS MiddleWare) interface [5], allowing portability among DDS vendores, which consequently enables ROS developers to interpolate DDS implementations, based on their application's needs during runtime. It should be noted that the DDS implementations are low-level of abstraction, strictly defined by its corresponding vendor's API. DDS only defines fundamental procedures at a higher degree of abstraction.

As the *rcl* layer works under ROS structures, to keep the passing of information through the middleware layer, a structure mapping between ROS and DDS must occur in the rmw interface, accounting the DDS implementation that is being considered at runtime. [5] The data structure mapping between ROS and DDS must account the preservation of the former ROS messages' structure. The *rcl* layer works under the ROS messages' `.msg` files so, these messages must be converted into messages' `.idl` files, to be handled by the DDS's RTPS wire protocol. [40] Then, the reverse process must be ensured as well, to fully complete the information flow, where the data must be converted back to ROS structures before being returned to the ROS client library.

2.2.3 *Network Concepts*

From a logical perspective [5], ROS applications are composed of many software modules that operate as separate computing processes, known as nodes, allowing its participation into the ROS global data space, similiar to a DDS participant. The data distribution between nodes is based on a publish-subscribe model, in which each node is perceived either as a publisher or as a subscriber within the network. First, nodes must be thoroughly covered before evaluating how data is passed and treated by publishing and subscribing.

Nodes can be perceived as processes that will likely perform computation over the network. The network is comprised of many nodes, running simultaneously and exchanging data between them, where each node addresses its corresponding module purpose. In a nutshell, nodes are network executables, built over a package, where their software code treats data that has been exchanged to each of them.

For instance, lets considered a well-known example called `TurtleSim`, which is a simulator typically used for learning ROS, mainly composed by 2 nodes, that perform together towards moving a turtle. Additional nodes were implemented, in order to add complexity to the current network, as to later support security matters as a proper example.

The ROS2 command toolset allows node launching using the `run` keyword, in which both the node's namespace and its corresponding package must be passed as arguments. It is worth mentioning that, nodes can be connected within a single package or between multiple packages, as they are built over their corresponding packages.

```
/multiplexer/multiplexer
/random/random
/turtle_teleop_key/turtle_teleop_key
/turtlesim_node/turtlesim_node
```

The `TurtleSim` node list.

The communication between nodes is done by message-passing, where the nodes involved in a particular communication exchange information using the *message* definition. A message is defined by its data type, also known as its *interface*, which can either be primitive (integer, string, boolean), or defined by a complex data structure, where multiple data types are assigned to their corresponding variables.

Each type of communication has its different corresponding interface, and they must be specified in different folders with unique namespaces. The folder namespace used for topic messages is `msg/`, whereas for service messages is `srv/` and for action messages is `action/`. Considering the `Turtlesim` example, the main message that is used to trigger the turtle movement, is composed by multiple data movement variables regarding the linear and circular movement of the turtle. The interface is called `Twist` and it is a ROS predefined geometry message.

```
Vector3   linear
Vector3   angular
```

`Twist.msg` interface file that is used to trigger the turtle movement.

Messages are passed through via the pattern of publishing and subscribing over a *topic*. A node publishes the message to any number of topics, that are then subscribed by nodes that want to get access to that message. Topics provide a multicast routing scheme, where publish data is casted into the multiple nodes that are subscribed to the topic.

A specific *topic* is created upon specifying its namespace over either a publisher or a subscriber callback instance, associated to a node executable. Message processing is done via the node's callback functions, which are activated upon message receipt, as it can also be utilized for publishing purposes. [5]

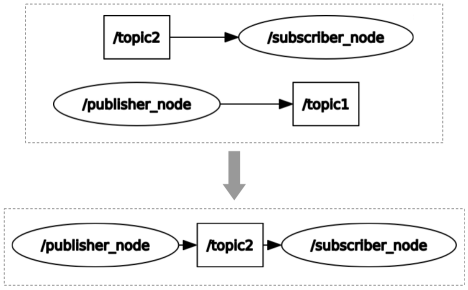

Figure 5: ROS2 communication behaviour over *topics*.

Addressing data exchange through a topic over nodes implemented in the same package, the configuration is straightforward, in which their respective subscriber and publisher instances must have the same topic's namespace instantiated. Whereas, if the nodes are specified in different packages, a technique of *remapping* must be used.

```
/multiplexer/high_topic
/multiplexer/low_topic
/multiplexer/main_topic
```

The `TurtleSim` topic list.

The topic list related to the `TurtleSim` example is depicted above. The 3 listed topics are vital for the communication process, and they all correspond to the `multiplexer` node. This happens because remapping between topic names is used to establish communication between packages, as explained above.

Both launching and remapping process are done by running a ROS *launch file*, that simultaneously launches each node, providing a simpler way of monitoring the system nodes. Every node is instantiated in this file, and have each topic remapping defined in `remappings` section. Also, distinctive namespaces allow the system to start the nodes, without any name nor topic name conflicts. However this technique has some flaws attached, since it does not furnish a way of launching nodes in a separated terminal, needed for user interaction purposes, like input reading.

By running the respective ROS `launch` command, will automatically compile and run the launch file related to the `TurtleSim` example. Every network node is now running simultaneously in the background. The only foreground running node is the turtlesim one, because it has an associated interface showing the turtle movement.

```
➜ launch ros2 launch turtle_launch.py
[INFO] [launch]: All log files can be found below /home/luis/.ros/log/2021-11-16-15-04-52-242468-mariolas-8851
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [random-1]: process started with pid [8852]
[INFO] [multiplexer-2]: process started with pid [8854]
[INFO] [turtlesim_node-3]: process started with pid [8856]
[INFO] [turtle_teleop_key-4]: process started with pid [8858]
```

Figure 6: Nodes launched after running the launch file.

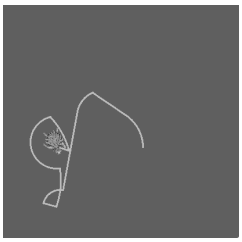

Figure 7: Turtlesim node interface.

Figure 8: Launching the `TurtleSim` network using a predefined launch file.

For understanding reasons, the reader may want to see how the network architecture is organized. ROS2 provides a GUI tool called *rqt*, that assists programmers in manipulating the network elements, in a more user-friendly manner. The graph visualizer, provided by this tool, allows the programmer to observe the network graph and how the elements are connected. While the nodes are running in the background, the reader may want to run the following command `rqt_graph`, in a separated terminal.

Figure 9: `TurtleSim` network graph presented by `rqt_graph`.

The `multiplexer` has two different active subscriptions, managing two different turtle movement values, it must keep them synchronized with each other. This is achieved by setting different priorities to each subscription, where the keyboard node has priority over the random controller, meaning that when the keyboard publishes movement data through its corresponding topic, the `multiplexer` must process that received data.

Timers are also used, since they provide a useful way of managing these topics, by time-assigning, alternately changing the priority after the timer runs out, through its corresponding callback function.

Given the priority set at each moment, the `multiplexer` has to forward the movement command, either from the keyboard or the random controller, through the `main_topic`, enabling the turtle movement. Every aspect related to the publisher-subscriber pattern, in this ROS2 system, is treated over topic's namespace remapping.

Additional ROS network elements

The `TurtleSim` application given as example above, does not address all the existing ways of data-exchanging between nodes. Even though *topics* are the most conventional way of communication, due to its multicast scheme, subscribers can not be identified by the publishers, so logging and synchronization becomes rather difficult.

SERVICES *Services* allows a client, that can also be seen as a topic subscriber, to request data from a server, that likewise a topic publisher, furnish data through a service. The data is only provided when the client node makes a request. Each service is always linked to just one server node, and does not maintain active connections. To address the service stalling that the former ROS issued due to the service's synchronization nature, ROS2 services are asynchronous, since it is possible to specify a callback function that is triggered when the service server responds back to the client.

ACTIONS Other notable way of exchanging data is by setting goals through *Actions*. Actions, likewise services, also uses a client-server model, but they were design for other purposes rather than only processing

a request and sending back a response. Actions are intended to process long-running tasks, where the client sends a goal request to the server node, that confirms the receiving of this goal. Before returning a response back to the client, the server can send feedback back to the client. Unlike services, actions can be cancelled, so the return response could not be acknowledge. A worth mentioned example, commonly used in robotic systems for navigation, is the intent of moving a robot to a position, previously requested by a client. While its traveling, it can send information about the transition state. When the robot reaches the predefined position, the server acknowledges the client by sending a result message.

PARAMETERS Another relevant concept behind ROS is the existence of nodes *parameters*, that allows individual configuration of the network nodes. In the former version of ROS, the node parameters were controlled by a global *parameter server*, managed by its corresponding ROS Master. However, in ROS2 each node declares and manages its own parameters, by using the predefined commands `get` and `set`. Additionally, using a parameter function callback, the node's parameters can easily be edited.

NODE COMPOSITION Usually a node is attached to a single process, but it is possible to combine multiple nodes into a single process, structurally abstracting some network parts, while improving the network's performance. However, there is a slight difference about how ROS and ROS2 approaches the node composition. In the former version of ROS, node composition was done over the combination of *nodelets*, intentionally designed to ease the cost of overusing TCP for message-passing between nodes. Supported by the former idea of *nodelets*, ROS2 introduces the *components* as software code compiled into shared libraries, that can be loaded into a *component container* process at runtime in the network, ensuring node composition. Node composition could also be applied for security matters. Suppose a scenario where multiple nodes respect the same security policies. By combining them into a single process, the mapping into this set of rules would be direct, easing the usage of security enclaves.

2.3 SECURITY INTEGRATION

The concerns about security take on particular importance, since robots are becoming more frequently used and can directly affect the physical world. *Cyber-physical* systems, commonly related to the arising of the automatizing concept of robots, feature unique vulnerabilities that exploit both cyber and physical nature of these devices, mainly by sensing and actuating. Sensing becomes relevant when it comes to the potential leak of information about the surroundings of its robot. Whereas, actuating is related to the functionality of the robot, where physical safety is the main concern. [28]

As aforementioned, ROS middleware faces known vulnerabilities due to its architecture and model nature. ROS internal communication is built around TCP ports, allowing robots to be built as distributed modules. As a result techniques such as port scanning is usually used to compromise the data itself. Due to the ROS master role in the communication architecture, and its ability to connect to other nodes, exposing this node using port scanning poses a threat over the whole network. [10]

There is also worry regarding how ROS controls node communication, as node-to-node communications are settled in plain text. Despite the fact that this method of handling communications has several advantages, namely improved performance and data recovery, it is evident that data content is vulnerable to unauthorized usage.

The DDS integration over ROS2 besides ensuring a well-structured communication architecture, also guarantees security properties, where ROS's flexibility is preserved despite this increase on security. The *DDS-Security* [31] is a specification that serves as an add-on to the DDS protocol, defined by a set of plugins (Authentication, Access Control, Cryptographic, Logging, Data Tagging), combined in a Service Plugin Interface (SPI) architecture. The provided specification ensures protection against unauthorized access to data, considering the publish-subscribe pattern, as well as data manipulation and replay attacks. [2, 17]

Depending on the security requirements needed for a particular application, the usage of these plugins may be adjusted. The existence of a *Standardized Plugin API* enables this customization, serving as an interface between modules and DDS security protocols, while also incorporating all features of secure communications.

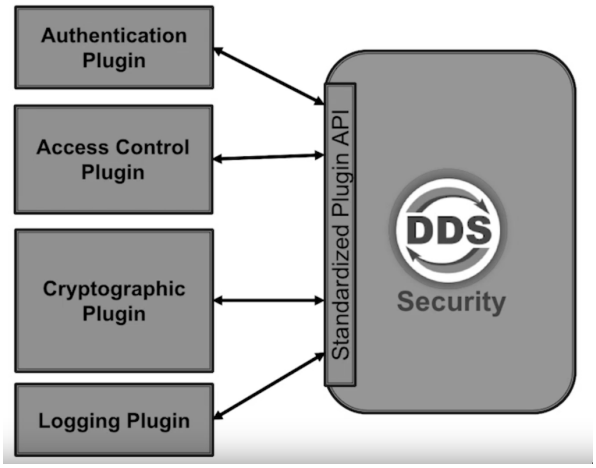

Figure 10: DDS-Security API with their respective security plugins.

Every DDS implementation supported by ROS2 makes use of the DDS-Security specification, enabling security over the ROS application environment. Even though ROS2 is deployed without security mechanisms by default [17], ROS2 provides a toolset, the Secure Robot Operating System 2 (SROS2) toolset, concerning the DDS-Security functionality.

The control over these tools are done by the ROS client library, providing security over the Application layer, while DDS is capable of providing security over the communication architecture. Each domain participant in ROS2 has a set of associated security files and two different security approaches (permissive and strict) are supported features considered by the SROS2 toolset.

Since this security integrity on ROS2 is consider a recent technology implementation [17], the developer must be aware of improper configuration, that can still lead to security problems. However, the variety of capabilities in the offered SROS2 toolset attempts to aid with security configuration across environments.

2.3.1 *Security Analysis*

To properly introduce the set of tools that SROS2 provides, the `TurtleSim` application already presented will now account the security features, as to provide authentication and encryption over the network communication, as well as access control policies over the application nodes.

The `multiplexer` node handles commands related to turtle's movement, acting as a topic selector between two different subscribed topics, each of them was respectively associated with a priority value. Based on the priority valued, the `multiplexer` node forwards the commands, related to the selected topic, into the `turtlesim` node, triggering the turtle's movement.

However, `multiplexer` is not exclusive to the `turtlesim` node, as it is still possible to directly publish commands to the topic that handles the turtle's movement, since security policies are yet to be implemented.

To accurately achieve this exclusivity, in which the turtle's movement is uniquely concerned by the `multiplexer` node, *access control* policies must be applied. The remaining nodes should be considered untrustworthy, denying any potential undesired turtle's movement. The idea is to encapsulate both `multiplexer` and `turtlesim` nodes, as the `multiplexer` monitors and manages all data intended to manipulate turtle's movement.

In technically terms, a *keystore* must be initiated, as SROS2 yields a command that permits its creation. A keystore is a created directory where files regarding security are stored. By generating a keystore directory, it may then be sourced and utilized by *rcl* features towards applying security to the application.

```
ros2 security create_keystore turtlesim_keystore
```

Keystore creation using the proper SROS2 command.

The `security` command enables features regarding security matters, concerning the DDS-security artifacts.

Keystore's Directory Architecture

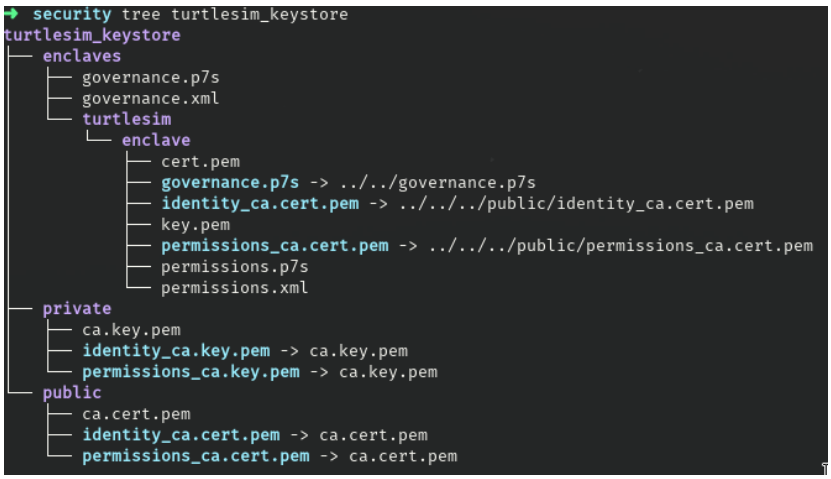

Figure 11: `Turtlesim_keystore`'s directory structure.

- The `enclaves` directory contains the security tools related to each enclave created. An **enclave** is a group of ROS nodes, controlled by the same set of security rules, defined in its corresponding enclave directory. Each enclave includes files needed to enable security, such as CA certificates and their own private key (`key.pem`). Besides containing each enclave created (for instance, in the figure above, a `/turtlesim/enclave` enclave is created), this directory also has a governance policy document `governance.xml`, as well as a signed copy file related to the CA permissions, `governance.p7s`.

- The `public` directory contains material that is permissible as public. A Certificate Authority certificate, `ca.cert.pem` is stored in this directory. This file stores the CA **public key** and it is used to validate the identity and permissions of each ROS network node by the CA. Note the existence of `identity` and `permissions` folders with their respective certificate/key file. This is due to the fact that DDS supports the separation of identity and permission chains, however ROS usually uses the former `ca.cert.perm` file, meaning that only a CA is used for both these processes.

- The `private` directory contains material that is considered private. A Certificate Authority certificate, `ca.cert.pem` is stored in this directory. This file stores the CA **private key**. It is used to modify the network policies, such as access permissions, and to add new participants. Similar to the `public` directory, the CA key corresponding to its identity and permissions can be stored in their corresponding individual directories.

Understanding Security Enclaves

Once the keystore has been created, the respective enclaves can be implemented. As mentioned, an enclave is a group of nodes that follow the same security policy. As enclaves are specified upon execution time, their security artifacts are actually used by running processes.

Since an enclave is a DDS domain term, typically a node is an abstraction of a DDS `participant`. However, by considering node composition, as a reliable way for matching multiple nodes simultaneously to the same enclave, this node perception as participants can not be taken into account, due to causing non-negligible overhead. There is also not convenient to compose nodes as individual participants, as far as operating system's security is concerned, where permission distribution and memory becomes rather difficult to handle.

To adress this, each participant must be matched to a node shared context, instead of being directly related to a specific node. Thereby, the initial given definition of an enclave is not totally correct, since a participant can either be perceived as single node or as multiple node shared context.

Each enclave security artifacts are used by its respective DDS participant. Meaning that nodes mapped into the same context are controlled by the same enclave's security policies.

```
ros2 security create_key turtlesim_keystore /security_enclave/
```

Creation of an `enclave` into the predefined keystore.

The following exports need to be sourced to force SROS2 security features, as they concern relevant environment variables.

The first sourced variable points to the keystore's root, allowing ROS2 to identify where the security artifacts are kept. The second regards serves as the security enabler. The last variable sets which security strategy will be used when dealing with security files.

```
export ROS_SECURITY_KEYSTORE=/path/to/the/created/keystore
export ROS_SECURITY_ENABLE=true
export ROS_SECURITY_STRATEGY=Enforce
```

SROS2 environment variables.

The keystore creation, alongside with their respective enclaves, only ensures security over the network communication, in which node authentication and data encryption are concerned. By using tools like `tcpdump`, data encryption can be easily verified. Authentication is ensured upon the enclave's creation. However, to properly apply security over the `TurtleSim` application, acess control policies must be appropriately covered.

Access Control

In order to provide access control, the permission files, of the enclave that is being considered, need to be modified. This is ensured by adding security permissions to these files, with the mandatory signature of the Certificate Authority.

However, a suitable way of editing the permission file, `permissions.xml` (file that dictates how the enclave manages the permissions within the network) is by creating a policy file and running the following command.

```
ros2 security create_permission turtlesim_keystore /enclave policy_file
```

Setting permissions into an enclave.

TurtleSim regarding Security

2.4 RELATED WORK

This section aims to present previous works that attended security matters related to the deployment of robotic systems using Robot Operating System as its application enhancer. When developing real-time systems, safety matters are often referred as critical, because of the overall integration with the real world. Security however, it is measured by evaluating different security issues by properly analysing the system model. However, due to the high nonlinearity and complexity of real-time systems, implementing such a thorough analysis method in near real-time remains a significant difficulty.[11]

The deployment of real-time systems results in the importance of concerning about safety in a performance point of view, resulting of the demanding time-critical scenarios. Many studies were made regarding the performance

in both ROS and ROS2 (also regarding DDS Quality of Service policies [27]), by analysing using performance measurement approaches, providing a guided and supported review on how performance can directly affect time critical situations, where safety is envolved.[27, 5]

System security concerning network exposure, often explored by unauthorized acess and data leaking, can be treacherous and it is considered a complex subject, due to the abundance of different network security technologies that do not cover every security aspect, since absolute security does not exist, as new vulnerabilites arise from the tecnology evolution.[21] The creation and deployment of security countermeasures are essential upon configurating the network towards achieving security. Within this vast topic, several different avenues of endeavor come to mind, each deserving of a substantial study. Network security means exploring the network beforehand by computer intrusion detection, traffic analysis, network monitoring, alongside many other practical networking security aspects.[26]

The literature concerning the network security enhancment that Robot Operating System 2 furnishes, by offering the SROS2 toolset, is quite limited. Most of the existing work is on the exploration of the former version of ROS in terms of port exposure, contextualized in the approach considered to protect the system network.

Many researches were made regarding this issue that ROS faces, one in particular that explored the IPv4 address space of the Internet for instances of ROS, named *Scanning the Internet for ROS: A View of Security in Robotics Research*[10], with the goal of identifying ROS vulnerable hosts, mostly master nodes since they provide information about their related topics and node's parameters, mainly by port scanning, so that developers could be aware of the possibility of exposure of their robots. The performed scans furnished information about hosts that could either be a sensor, an actuator or even a simulator. Topics were also identified since they provide evidence of what is likely to be available to an attacker. This study is rather relevant because of how easily can attackers gather information about potential robots, and control them further on, through the public Internet, making it unavoidable to develop mechanisms concerning security.

A different approach was previously presented[12] when it comes to protecting the ROS network, which primarily focused on applying security measures on the application layer, treating ROS architecture as a black-box, by mainly running an Authentication Server, storing certificates and files related to trusted domain participants, while controlling and providing session keys related to the communication process. Even though the message data can not be perceived, unauthorized nodes can still collect information about the flow of the data, since ROS architecture is responsible for the publish-subscribe interaction, meaning that security issues still persist which cannot be handled on the application level alone.

Secure Robot Operating System (SROS)[39] was initially developed as an experimental tool (later evolved to SROS2 as a supporting tool for ROS2), which supports TLS for all socket transport, node restrictions and chains of trust, guaranteeing publishers authorization when it comes to publish to a specific topic. Another worth-mentioned tool is Rosbridge[8], which provides a WebSocket interface to ROS and corresponding server to allow interaction between applications and ROS nodes, by using TLS as support and also access control over topics and API calls.

3

ALLOY SPECIFICATION FRAMEWORK

As aforementioned, the main problems that this thesis aims to tackle are security vulnerabilities and data processing over big workloads of streaming data. In this chapter, it is intended to explore technologies that are relevant to overcome the above-mentioned challenges (**??**) as well as previous developed work that has the same or similar goals as this thesis (**??**).

3.1 BACKGROUND

Enterprises, particularly startups and small and medium businesses, are gradually shifting to the Cloud to outsource data and processing. [**?**] As advantageous as this may be, the sharing of content with third-party systems, poses challenges about privacy and security. As previously stated (**??**), traditional strategies for achieving confidentiality, such as a user encrypting their data with their own key or even using techniques such as homomorphic encryption, make some computations unfeasible, hence alternative methods must be investigated.

3.2 RELATED WORK

4

CURRENT WORK

5

CONCLUSION

BIBLIOGRAPHY

[1] Alaa Alaerjan, Dae-Kyoo Kim, and Dhrgam Al Kafaf. Modeling functional behaviors of dds. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1–7. IEEE, 2017.

[2] Kai Beckman and Jonas Reininger. Adaptation of the dds security standard for resource-constrained sensor networks. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–4, 2018.

[3] Dirk Beyer and Thomas Lemberger. Software verification: Testing vs. model checking. In *Haifa Verification Conference*, pages 99–114. Springer, 2017.

[4] Renato Carvalho, Alcino Cunha, Nuno Macedo, and André Santos. Verification of system-wide safety properties of ros applications. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7249–7254, 2020.

[5] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B Brandenburg. Response-time analysis of ros 2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[6] Steve Cousins. Exponential growth of ros. *IEEE Robotics & Automation Magazine*, 1(18):19–20, 2011.

[7] John J Craig. Introduction to robotics. 2005.

[8] Christopher Crick, Graylin Jay, Sarah Osentoski, Benjamin Pitzer, and Odest Chadwicke Jenkins. Rosbridge: Ros for non-ros users. In *Robotics Research*, pages 493–504. Springer, 2017.

[9] Fast DDS. Dds architecture. https://fast-dds.docs.eprosima.com/en/latest/fastdds/getting_started/definitions.html, 2019.

[10] Nicholas DeMarinis, Stefanie Tellex, Vasileios P. Kemerlis, George Konidaris, and Rodrigo Fonseca. Scanning the internet for ros: A view of security in robotics research. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8514–8521, 2019.

[11] Ruisheng Diao, Vijay Vittal, and Naim Logic. Design of a real-time security assessment tool for situational awareness enhancement in modern power systems. *IEEE Transactions on Power systems*, 25(2):957–965, 2009.

[12] Bernhard Dieber, Severin Kacianka, Stefan Rass, and Peter Schartner. Application-level security for ros-based applications. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4477–4482, 2016.

[13] Vincenzo DiLuoffo, William R Michalson, and Berk Sunar. Robot operating system 2: The need for a holistic security approach to robotic architectures. *International Journal of Advanced Robotic Systems*, 15(3):1729881418770011, 2018.

[14] ROS 2 Documentation. Tcpros. http://wiki.ros.org/ROS/TCPROS, 2013.

[15] ROS 2 Documentation. Udpros. http://wiki.ros.org/ROS/UDPROS, 2013.

[16] euRobotics. Why is robotics important. https://www.onlynaturalenergy.com/why-is-robotics-important/, 2021.

[17] Kyle Fazzari. Ros 2 dds-security integration. https://design.ros2.org/articles/ros2_dds_security.html, 2020.

[18] DDS Foundation. About dds. https://www.dds-foundation.org/what-is-dds-3/, 2021.

[19] Software Design Group. Alloy 6. https://alloytools.org/alloy6.html, 2021.

[20] Daniel Jackson. Alloy: a language and tool for exploring software designs. 62:66–76, 2019.

[21] Merike Kaeo. *Designing network security*. Cisco Press, 2004.

[22] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296, 2018.

[23] Jongkil Kim, Jonathon M Smereka, Calvin Cheung, Surya Nepal, and Marthie Grobler. Security and performance considerations in ros 2: A balancing act. *arXiv preprint arXiv:1809.09566*, 2018.

[24] Man Lin, Li Xu, Laurence T Yang, Xiao Qin, Nenggan Zheng, Zhaohui Wu, and Meikang Qiu. Static security optimization for real-time systems. *IEEE Transactions on Industrial Informatics*, 5(1):22–37, 2009.

[25] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. page 373–383, 2016.

[26] Gerald A Marin. Network security basics. *IEEE security & privacy*, 3(6):68–72, 2005.

[27] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. In *Proceedings of the 13th International Conference on Embedded Software*, pages 1–10, 2016.

[28] Jarrod McClean, Christopher Stull, Charles Farrar, and David Mascarenas. A preliminary cyber-physical security assessment of the robot operating system (ros). In *Unmanned Systems Technology XV*, volume 8741, page 874110. International Society for Optics and Photonics, 2013.

[29] Nader Mohamed, Jameela Al-Jaroodi, and Imad Jawhar. Middleware for robotics: A survey. In *2008 IEEE Conference on Robotics, Automation and Mechatronics*, pages 736–742. Ieee, 2008.

[30] Object Management Group (OMG). Data distribution service (dds). https://www.omg.org/omg-dds-portal/.

[31] Object Management Group (OMG). Dds security. https://www.omg.org/spec/DDS-SECURITY/1.1/, 2018.

[32] Gerardo Pardo-Castellote, Bert Farabaugh, and Rick Warren. An introduction to dds and data-centric communications. *Real-Time Innovations*, 2005.

[33] Jon Postel et al. User datagram protocol. 1980.

[34] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[35] Open Robotics. The robot operating system. https://www.ros.org.

[36] Open Robotics. Ros 2 documentation. http://docs.ros.org/, 2021.

[37] OMG Specification. The real-time publish-subscribe protocol (rtps) dds interoperability wire protocol specification. *Object Manag Group Pct07-08-04*, 2007.

[38] George Stavrinos. Client libraries. http://wiki.ros.org/Client%20Libraries, 2020.

[39] Ruffin White, Dr Christensen, I Henrik, Dr Quigley, et al. Sros: Securing ros over the wire, in the graph, and through the kernel. *arXiv preprint arXiv:1611.07060*, 2016.

[40] William Woodall. Ros 2 on dds. https://design.ros2.org/articles/ros_on_dds.html, 2019.

[41] Seongjin Yun, Jun-Hong Park, and Won-Tae Kim. Data-centric middleware based digital twin platform for dependable cyber-physical systems. In *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 922–926. IEEE, 2017.

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.