ROSRV: Runtime Verification for Robots

Jeff Huang[1], Cansu Erdogan[1], Yi Zhang[1], Brandon Moore[1], Qingzhou Luo[1], Aravind Sundaresan[2], and Grigore Rosu[1]

[1] University of Illinois at Urbana-Champaign
[2] SRI International, Menlo Park, CA 94025
{smhuang,bmmoore,cerdoga2,qluo2,grosu}@illinois.edu
aravind@ai.sri.com

Abstract. We present ROSRV, a runtime verification framework for robotic applications on top of the Robot Operating System (ROS [8]), a widely used open-source framework for robot software development. ROSRV aims to address the safety and security issues of robots by providing a transparent monitoring infrastructure that intercepts and monitors the commands and messages passing through the system. Safety and security properties can be defined in a formal specification language, and are ensured by automatically generated monitors. ROSRV integrates seamlessly with ROS—no change in ROS nor the application code is needed. ROSRV has been applied and evaluated on a commercial robot.

1 Introduction

The Robot Operating System (ROS [4]) is an open-source meta-operating system for robot software development. With the increasing popularity of programmable robots, ROS has become the de facto standard for robotic applications such as perception [3], motion planning [1], object detection [2], etc. ROS provides common robot-specific libraries as well as standard operating system services such as hardware abstraction, low-level device control, etc[3]. At the lowest level, ROS offers a message passing interface that provides inter-process communication including publish/subscribe messages and distributed parameter configuration. The message passing is based on a graph architecture where computation takes place in ROS processes (called *nodes*) that may receive, post and multiplex messages.

With the wide adoption of ROS, however, its safety and security are becoming an important concern. For instance, any node is allowed to publish/subscribe arbitrary messages on any topic[4], which can be easily abused by attackers. Moreover, ROS is designed to be highly dynamic and distributed, making it hard or impossible to verify statically. For example, nodes running on different hardware devices can join and leave dynamically, changing parameters and namespaces,

[3] Note that ROS is not a traditional operating system. For example, it does not deal with process scheduling.

[4] In ROS, topics are communication channels between publishers and subscribers which identify the content of the message.

and creating new message topics. A node can be killed by another node via a shutdown command or accidentally replaced by a new node with the same name.

In this paper, we present a runtime verification framework, ROSRV, for improving the safety and security of robots running ROS. ROSRV is designed to be lightweight, expressive, and transparent, with no changes to ROS or the application running on top of ROS. Its core is a runtime monitoring infrastructure that intercepts, observes and optionally modifies the commands and messages passing through the system, and performs actions upon relevant events defined over the messages. Safety properties are implemented as monitors on top of this infrastructure, such that all relevant messages are monitored and property-triggered user-specified actions are performed. For example, to prevent a robot from overturn, a safety monitor can intercept and modify the robot speed/acceleration messages. ROSRV provides a specification language for safety properties, and monitors are automatically generated from specifications. For security, ROSRV provides a specification language for access control policies and enforces them at runtime. For example, it is possible to specify which nodes are permitted to publish messages on certain topics or to send shutdown signals to kill other nodes.

ROSRV integrates seamlessly with ROS and has been applied on the LandShark[5], an unmanned ground vehicle (UGV) robot running ROS. We illustrate ROSRV via a case study on LandShark and demonstrate how ROSRV improves the safety and security of LandShark through monitoring safety properties and enforcing access control policies. All the ROSRV source code, materials, and demos are publicly available at `http://fsl.cs.illinois.edu/rosrv`.

2 Robot Operating System (ROS)

We briefly overview ROS communication concepts [7], highlighting its safety and security limitations. ROS is a peer-to-peer network of nodes that communicate with each other using XMLRPC and custom ROS messages that are based on TCP/IP. Each message has a type and is transported on a channel called a *topic*, and each node may subscribe and publish to arbitrarily many topics. A special node, the *Master*, coordinates the communication and provides global services such as naming, registration, parameter updates and lookups.

Fig. 1 depicts the ROS communication architecture. Node communication is initiated with a sequence of XMLRPC requests. First, nodes register with the Master; e.g., the publisher may register that it publishes messages on topic "chat" at address "foo:1234", and then when a subscriber registers to topic "chat" the Master passes it the publisher's address. Second, the subscriber contacts the publisher to obtain a topic connection and negotiate the transport protocol. Finally, the subscriber connects to the publisher and starts receiving messages. ROS also supports commands that query/update the system state, such as the name, address, and published/subscribed topics of a node, query published topics, kill a node, etc.

[5] The LandShark UGV is a product of Black-i Robotics (`www.blackirobotics.com`).

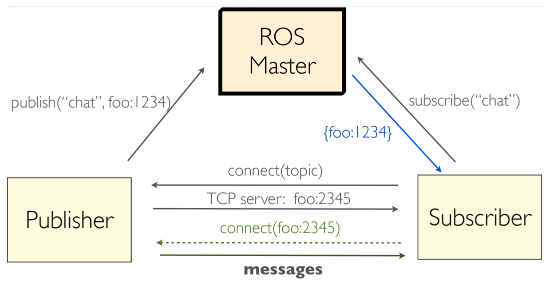

Fig. 1. ROS Communication Architecture

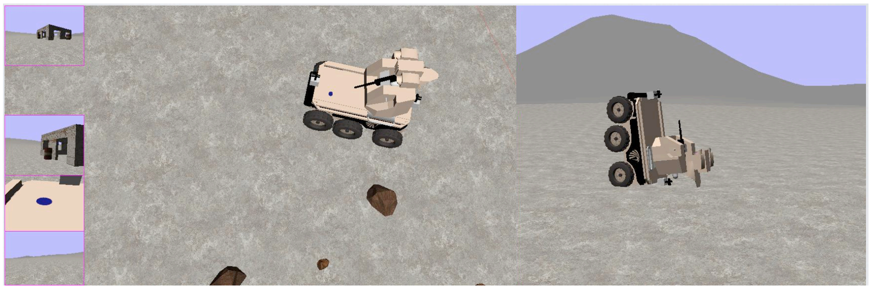

Fig. 2. Safety issues: LandShark shoots itself and tips over

2.1 ROS Safety and Security Issues

Safety is often application specific and challenging to address in a generic distributed communication framework such as ROS. Also, since ROS is designed to be open and dynamic, it lacks any security protection mechanism. For example, an attacker can easily create a node, query the Master about the system state, and send shutdown commands to kill any node. Moreover, nodes are uniquely identified by their name, with newly created nodes replacing existing ones with the same name. Thus, an attacker can easily fake a node to publish bogus messages on important topics. For instance, a navigation node in a robot may be killed and replaced by a fake node that misdirects the robot.

The LandShark UGV has an onboard Linux box connected to various devices: GPS, radar, cameras, motor and turret controllers, a paintball gun, etc. The paintball gun can fire on receiving a trigger message. Each device has a driver and a corresponding ROS node (wrapper) which publishes sensor data and/or accepts commands as ROS messages. An operator control unit (OCU) node listens to messages from the robot and sends it user commands.

Fig. 2 shows two scenarios where the safety of the robot is infringed. The first, "robot shoots itself", is motivated by the fact that no mechanism in the LandShark or ROS prevents this behavior, so the LandShark can shoot itself (inadvertently or maliciously, by an attacker). The second, "LandShark tips over", occurs when it accelerates too quickly or becomes unbalanced. This scenario is typical for UGVs but there is no safety mechanism in ROS to prevent it.

3 ROSRV

ROSRV is designed to address the safety and security issues in ROS-based robot applications. Fig. 3 shows its architectural overview. The main difference from ROS is the *RVMaster* node, which acts as both a secure layer protecting ROS Master and as a functional layer for protecting the safety of the application: all node requests to ROS Master can be intercepted by RVMaster and all messages can be monitored, and thus the desirable safety and security policies enforced.

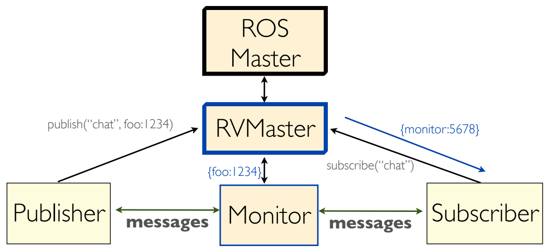

Fig. 3. ROSRV Architecture

For example, access control policies such as allowing only certain nodes to publish messages on a certain topic can be enforced by checking the node identity and topic name in the request and rejecting it if they do not match the policy. We have developed an IP address-based access control specification that allows the security policies to be enforced as system configuration.

Monitors are generated by RVMaster from safety specifications and implemented as ordinary ROS nodes that can subscribe and publish messages. However, RVMaster keeps track of all the communication requests by the other nodes in the system and manipulates the point-to-point communication addresses so that the generated monitors act as men-in-the-middle. For example, a monitor can drop the triggering message of LandShark when the position of the gun is within the range of pointing at itself, or modify the acceleration message when the LandShark is moving too fast to prevent it from tipping.

An important property of ROSRV is that it does not require any change to ROS or the application code. The only requirement is to configure the RVMaster to listen at the standard port and the ROS Master to listen at a hidden port visible only to RVMaster. This is implemented using a firewall to block access to the ROS master port. In this way, all the ordinary nodes in the system remain the same (sending requests to the default port, sending and receiving messages from normal ROS nodes), and are not even aware of being monitored.

3.1 Monitoring Safety Properties

A safety property is specified by means of events and actions based on event sequences. Fig. 4 shows a simple example to illustrate the idea. The property we want to monitor here requires that the robot can only fire in certain safe poses. There are two events, `checkPosition` and `safeTrigger`. Each event has its own parameters and the topic and type of the messages being monitored. For example, `checkPosition` is used to check whether the gun is at a safe position: "`position > -0.45`" (not pointing at itself). It listens to topic */landshark/joint* *_states* with message type *sensor_msgs/JointState*, which holds two arrays: `name` and `position`. The second elements of these arrays are bound to variables `N` and `P`, respectively. They are used as the parameters of the event and in the event handler code. Event handler code is used to trigger actions under certain conditions. For example, in `checkPosition` the global variable `isSafeTrigger` will be set to true if and only if the gun is at an angle larger than `-0.45`; later on it is used in `safeTrigger` to determine whether the trigger is allowed or not.

```
safeTrigger() {
      bool isSafeTrigger = false;
      event checkPosition(string N, double P)
               /landshark/joint_states sensor_msgs/JointState
               ‘{name[1]:N, position[1]:P}’  {
      if (N=="turret_tilt"){if (P > -0.45){ //check gun position
                              isSafeTrigger = true;
                      }else{
                              isSafeTrigger = false;
      }      }     }
      event safeTrigger() /landshark_control/trigger
                landshark_msgs/PaintballTrigger ‘{}’  {
                if(!isSafeTrigger) return; //drop trigger message
      }
}
```

Fig. 4. Safe Trigger Specification

Our monitoring infrastructure enables us to use any *logic plugins* of Monitoring-Oriented Programming (MOP [5]) to specify temporal properties over events, such as regular expressions, linear temporal logics, context-free grammars, etc., and to trigger actions only when certain patterns of event sequences are matched, because such specifications translate into ordinary code that implements corresponding monitors, which are executed as event actions.

ROSRV automatically generates C++ monitoring code from all the user-defined specifications, and creates nodes that act as monitors as explained above. Each event generates one call back method and all the call back methods are registered by RVMaster. Parameters of events are treated as references to fields in monitored messages, so users can modify messages in event handler code. Event handlers are inserted in call back methods that are called at runtime.

3.2 Enforcing Security Policies

ROSRV enforces access control based on a user-provided specification of access policies as input configuration. On receiving any XMLRPC request, RVMaster decides whether the request is allowed to go to the ROS Master according the specification. The policies are currently categorized into four different sections: [Nodes], [Subscribers], [Publishers], and [Commands]. Under each section, the access policy is written as a key followed by an assignment symbol and a list of values. For [Nodes], “key” is the node name, and “value” is the machine identity allowed to create nodes with the name “key”. For [Subscribers] and [Publishers], “key” is the topic name and “value” is the node identity allowed to subscribe/publish to the topic. For [Commands], “key” is the command name and “value” is the node identity allowed to perform the command. We support access granularity at a host level. We use the source IP address of the request

```
[Groups]
localhost = 127.0.0.1
certikos = ip1 ip2 ip3 ip4
ocu = ip5 ip6 ip7 ip8
[Nodes]
default=localhost
/landshark_radar=certikos
```

```
[Publishers]
default=localhost certikos
/landshark_control/trigger= ocu

[Subscribers]
default = localhost certikos
/landshark/gps = ocu
```

```
[Commands]
# Commands: full access
getSystemState = localhost certikos ocu
# Commands: limited access
lookupNode = localhost certikos
# Commands: local access only
shutdown = localhost
```

Fig. 5. Sample access control policy for LandShark

to identify the host, because the node name itself is self-reported. IP address aliases and groups are also supported in our specification language.

Fig. 5 shows a snippet of the LandShark access policy. The [Group] section defines three groups of IP addresses. In the [Nodes] section, "`default=localhost`" means that by default "`localhost`" is allowed to create a node with any name, and "`/landshark_radar=certikos`" that the alias "`certikos`" is allowed to create a node with name "`/landshark_radar`". In [Publishers], only nodes running on machine "`ocu`" can publish to topic "`/landshark_control/trigger`". In [Commands], "`getSystemState=localhost certikos ocu`" means that nodes running on machines "`localhost`", "`certikos`", or "`ocu`" are allowed to send "`getSystemState`" requests to ROS Master, and "`shutdown=localhost`" that only nodes on "`localhost`" are allowed to "`shutdown`" other nodes.

4 Current Limitations and Future Work

Security The main limitation of the current implementation is the reliance on IP addresses in particular and on network routing in general to guarantee security. Naively trusting IP addresses does not protect against attackers who can run processes on the same (virtual) machines as trusted nodes, or spoof packets on physical network segments carrying unencrypted traffic. To defend against local attacks, the RVMaster and the mutually distrustful nodes can be run on separate (virtual) machines. To protect against spoofed IP addresses, the machines can be configured to receive packets from other machines on distinct virtual network interfaces, with the link between interface and machine using encrypted tunnels or relying on the virtual machine monitor to provide private local connections. Then routing can be configured so that only packets from specific machines can claim recognized IP addresses, and also to prevent nodes from being accessed directly. We intend to augment RVMaster with tools to automate the creation and configuration of virtual machines to more easily provide this level of security.

Scalability Currently ROSRV is centralized. All the monitor nodes live in the same multithreaded process, and all communication in the system is monitored. We tested the performance of monitoring with 10+ nodes in Landshark. The message delay caused by monitoring is on small digits of milliseconds. Although this is acceptable in our current project, the centralized monitoring may face scalability problems with a large number of nodes. We plan to investigate decentralized mechanisms such as multimaster [6] to improve scalability. The multimaster approach also enriches the fault tolerance of the system, as the current ROS master is a single point of failure.

Formal verification Currently the runtime verified system is not formally verified. This would require a formal model of ROS itself, as well as proving that the generated monitors and glue code guarantee the desired global system properties. At the implementation level, this would consist of showing that RVMaster respects the given model of ROS and invokes the monitor code at correct times to impose monitoring, and developing tools to prove that monitor code generated from higher level specifications actually correctly monitors those specifications.

5 Conclusion

With our society increasingly depending on robots, the importance of their safe and secure operation cannot be overstated. This paper makes first steps towards the runtime verification of robot applications. Users provide formal safety and security specifications, and monitors are automatically generated and incorporated in the system to ensure the safety and security of robots.

Acknowledgement: This material is based on research sponsored by DARPA under agreement number FA8750-12-C-0284. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal. STOMP: Stochastic trajectory optimization for motion planning. In *ICRA*, pages 4569–4574. IEEE, 2011.
2. U. Klank, D. Carton, and M. Beetz. Transparent object detection and reconstruction on a mobile platform. In *ICRA*, pages 5971–5978. IEEE, 2011.
3. B. Pitzer, M. Styer, C. Bersch, C. DuHadway, and J. Becker. Towards perceptual shared autonomy for robotic mobile manipulation. In *ICRA*, pages 6245–6251. IEEE, 2011.
4. M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.
5. G. Roşu and F. Chen. Semantics and algorithms for parametric monitoring. *Logical Methods in Computer Science*, 8(1):1–47, 2012.
6. ROS contributors. ROS Multimaster. `http://wiki.ros.org/rocon_multimaster`. [Accessed 2014-04-25].
7. ROS contributors. ROS technical overview. `http://wiki.ros.org/ROS/Technical%20Overview`. [Accessed 2014-04-25].
8. ROS contributors. ROS.org. `http://wiki.ros.org`. [Accessed 2014-04-25].

A Demo Overview

The ROSRV tool consists of the RVMaster node written in C++, a monitor generator (called *rosmop*) written in Java and JavaCC, and a set of bash scripts to compile and start ROS. The tool works on Ubuntu 12.04 with ROS Groovy distribution release. The user simply describes the property or a set of properties using the monitor specification language and specifies the access control policy in a configuration file. Taking the property specifications as input, ROSRV first automatically generates all the monitors, with each monitor corresponding to one property. The user can run *rosmop* with either a single property specification or the directory containing a set of property specifications, to generate the monitors and compile the whole system.

In the accompanying video, we demonstrate the use of ROSRV with three monitors and four access control policies on the Landshark robot.

– *Safe Trigger*: we first show that by default the Landshark can shoot itself when we move the gun to point at itself. We then enable the safe trigger monitor and show the gun is no longer allowed to shoot when pointing at Landshark, but can still shoot when pointing at the ground.
– *Safe Zone*: this monitor monitors the location of Landshark against a zone, and ensures that once Landshark enters the zone it cannot move out. Within the zone, we also show Landshark is disallowed to shoot itself, to demonstrate that the tool can support multiple monitors working simultaneously.
– *Logging*: in many cases, users want to log messages in the robot to understand the runtime behavior of the system. We show a logging monitor that, once enabled, prints out the messages that the user is interested in, which could be useful for debugging.
– *Access control policies*: we demonstrate a policy for each of the four sections in the sample access control policies in Fig. 5. We run this demo with two machines: one running ROSRV and all the legal nodes, and the other running the attacker nodes. We show the attacker nodes cannot perform any action not specified in the access control policy file, such as publishing messages on a certain topic, killing another node, or pretending to be an existing legal node.

The demonstration video is available at:
`http://fsl.cs.illinois.edu/index.php/ROSRV`