ROS-FM: Fast Monitoring for the Robotic Operating System(ROS)

Sean Rivera, Antonio Ken Iannillo, Sofiane Lagraa, Clément Joly, Radu State
SnT, University of Luxembourg
firstname.lastname@uni.lu

***Abstract*—In this paper, we leverage the newly integrated extended Berkely Packet Filters (eBPF) and eXpress Data Path (XDP) to build *ROS-FM*, a high-performance inline network-monitoring framework for ROS. We extend the framework with a security policy enforcement tool and distributed data visualization tool for ROS1 and ROS2 systems. We compare the overhead of this framework against the generic ROS monitoring tools, and we test the policy enforcement against existing ROS penetration testing tools to evaluate their effectiveness. We find that the network monitoring framework and the associated visualization tools outperform the existing ROS monitoring tools for all robots with more than 10 running processes and that the monitoring tool uses only 4% of the overhead of the generic tools for robots with 80 processes. We further demonstrate the effectiveness of the security tool against common attacks in both ROS1 and ROS2.**

I. Introduction

Robotics systems are usually larger systems of interconnected, distributed components. The role of robotics in society has continuously increased, spanning from industrial robots, consumer robots, commercial robots, to vehicles and drones. The market for robotics is ever-growing with a value of $115B in the year 2019 [28] and an expected CAGR of 25%.

To effectively monitor such robotic systems at run-time, special monitoring software is required. However, not many concepts and solutions have been proposed to monitor the components of robotic systems, especially in ROS (Robot Operating System). Security in ROS is an active concern, with few fully implemented solutions [13]. Any such monitoring systems have to be flexible, scalable, and secure, without sacrificing run-time. Currently, most monitoring software is processing intensive and slow, a double disadvantage on cyber-physical systems, particularly robotic systems. There are no current solutions that offer both monitoring and security without a severe processing impact.

In recent years, there have been many advancements in the field of process monitoring and analysis for Linux systems in the form of the extended Berkeley Packet Filter (eBPF) and the eXpress Data Path (XDP). These new technologies allow for the execution of limited software in kernel space at lower points in the data travel path, allowing for efficient and scalable processing. The processing capabilities of XDP and eBPF are utilized to develop *ROS-FM*, a new monitoring framework that provides a modular, scalable, and secure monitoring software for ROS that outperforms current solutions.

Problem statement How can we effectively monitor a robotic system without impacting performance? Can we secure the same system while staying within performance constraints?

Key challenges Designing a monitoring framework for ROS that answers these questions raises several challenges. Firstly, the framework must be easy to apply to a robotic system, as otherwise users will not be incentivized to use it (usability). Next, it must support all of the functionality that existing tools do, or it will be just a domain-specific tool (applicability). Additionally, it must be extensible to cover the large cyber-physical ecosystem without imposing major development costs (extensibility). Finally, it must be lightweight and efficient, as robotic systems operate in time-constrained environments on limited hardware (space and time efficiency).

In this paper, we demonstrate the benefits that eBPF and XDP can provide for robotic systems and propose a new system to leverage them. We build a new drop-in monitoring solution for ROS, a common robotic framework, which we then extend with two modules to show the full power available to developers.

The rest of the paper is organized as follows. Section II introduces and reviews the existing background in ROS and network framework. Section III provides the motivation and objectives of this paper. Section IV describes the related work. We detail our ROS monitoring tool in Section V. We then present the experimental results in Section VI. Section VII concludes and gives some future research.

II. Background

A. ROS

The Robot Operating System (ROS) [24], ROS1 by default, is a meta-operating system framework for developing robotic systems (see Figure 1) [27]. ROS provides independent computing processes called *nodes*, with the help of a master node, parameter server, and middleware layer. These systems allow nodes to communicate using *topics* and *services*. A more thorough analysis of the ROS framework is discussed by Rivera et al. [27].

ROS master and parameter server. The ROS core utility `roscore` contains the ROS master and parameter server and the node `rosout`. The master node tracks all the offered topics and services and maintains a map of the location of all nodes. Unfortunately, the master node is unable to enforce this map creating many vulnerabilities [14] [13] for ROS system. The parameter server acts as a global variable repository for

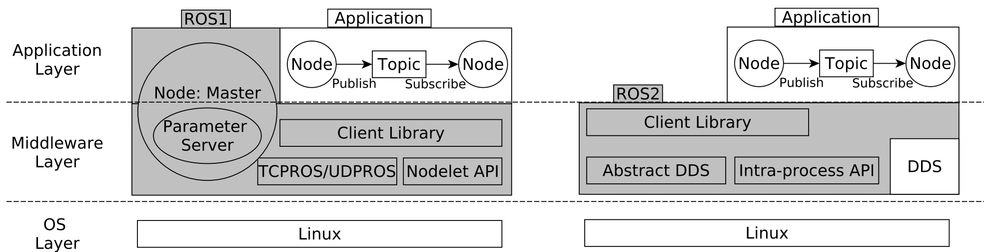


Fig. 1: ROS1/ROS2 architecture [24].

the nodes. The master node communicates through XMLRPC function calls, which carries several vulnerabilities.

ROS nodes. ROS uses nodes to define individual processes running within the system. Each node is designed to be self-contained and must follow a specific set of guidelines to integrate with the rest of the system.

Topics. Topics are the primary form of communication in ROS. Topics define the communication path with the ROS message framework by allowing nodes to join as either a publisher or a subscriber, to allow communication between nodes. To create a topic, a node informs the master node whether it is publishing or subscribing to a topic. If it is publishing, the exact port it will be publishing from is communicated. The master responds with a list of all publishing nodes (for subscribers) or all subscribers (for publishers), and then it notifies the remainder of the relevant nodes. The master server maintains a list of all topics and associated nodes and allows subscribers to connect to topics by opening a port. Anyone can publish any data on topics as there is no access control for topics beyond the data type MD5 hash.

Middleware layer. The middleware layer includes the communication system TCPROS/UDPROS. Each node must implement either TPCROS [25] and/or UDPROS [26]. TCPROS and UDPROS are extensions of the TCP and UDP protocols, with TCP being the preferred method with UDP being customizable by users. All TCPROS headers provide the name and an MD5 hash of the message type, utilizing the MD5 hash to ensure that the node is sending the correct type of message. Additionally, the initiator of a TCPROS connection must provide its name, the topic or service it wishes to connect to, and the expected message communication type. The ROS command-line interface allows interfacing with the robot system by launching processes as nodes.

Messages. Nodes communicate with each other using messages. Messages can hold a set of data that can be sent to another node via a typed field. Messages can be integers, floating points, Boolean, or custom by the developer.

ROS2. The second version of ROS, namely *ROS2*, is under development by the ROS community. The goals of ROS2 are to enhance the performance of multi-robot communications and provide real-time capabilities. The novelty in ROS2 is the modification of the middleware layer which uses the Data Distribution Service (DDS) as its networking middleware.ROS2 offers the following advantages regarding ROS1:

- Real-time requirements: initially designed for single robot control with no real-time and using reliable connections.
- Distribution: ROS1 uses a centralized discovery, the ROS master (single point of failure). In contrast, ROS2 is fully distributed including discovery.
- QoS: uses quality-of-service settings to handle lossy networks and efficient intra-process communication.
- New use cases operating in distributed environments: autonomous vehicles, multi-robot swarms,...etc.

However, as ROS2 is still under development, it is still unclear when a complete functional switch from ROS1 to ROS2 will occur. Presently there exists a helper node called *ros1-bridge* which allows developers to use hybrid systems of ROS1 and ROS2 to ease the transition. Thus, in this paper, we focus on problems, and corresponding solutions, of both ROS frameworks. For the rest of the paper, we use ROS to refer to both ROS1 and ROS2.

Typical ROS system. A typical ROS system is made up of 'multiple' packages (based on robot purpose) per robot [15], with each package containing an average of 3.8 nodes [30], and each node communicating on 5-8 topics [31]. These figures are measured on ROS1 but can be assumed to apply to ROS2 as well.

B. BPF/eBPF

The Berkeley Packet Filter (BPF) [20] is a high-performance limited instruction set for a bytecode virtual machine running inside the OS, to implement fast programmatic network filtering in kernel space. eBPF [3] is an extension to the classic BPF framework with hooks for generic event processing in the kernel, profiling programs, and libraries. This grants developers a greater number of processing capabilities for more complex applications.

eBPF allows a user-space application to inject code in the kernel at run-time. This means that the injection is performed without recompiling the kernel or installing any optional kernel module. The eBPF interpreter offers a limited selection of the standard C functionality, which allows the kernel to formally verify each section of the eBPF code. This limited selection does not allow loops and formally checks all memory accesses to ensure that the user-space code does not interfere with normal kernel functioning or the functioning of normal applications. Several kernel hooks are available for the network stack, such as the traffic classifier `tc` [10], or the eXpress

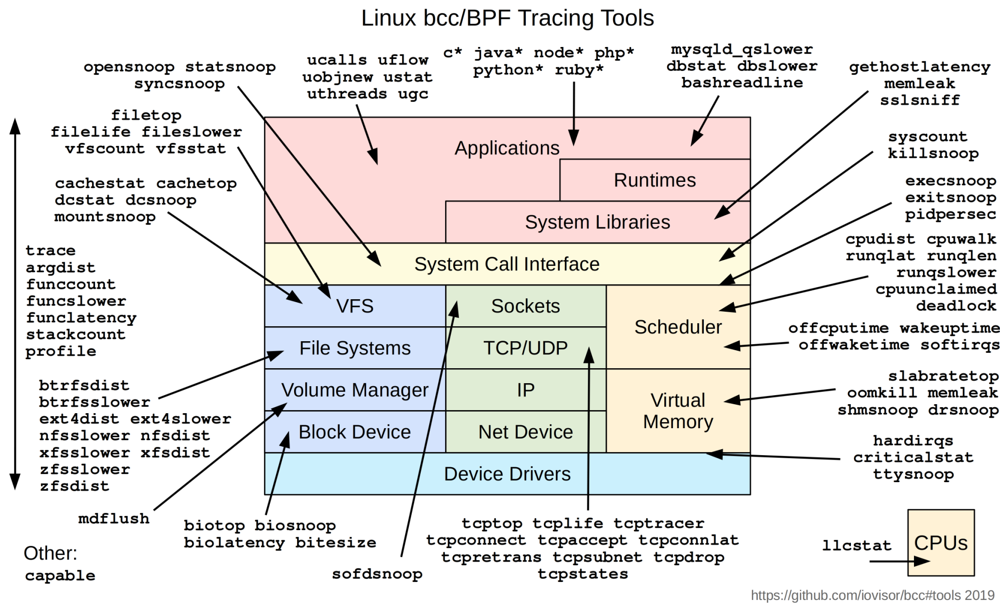


Fig. 2: All available bcc/BPF tracing and performance tools [18].

Data Path (XDP) [3], [16] (described in the next section), a low-level hook component executed before the network layer, used for DDoS mitigation [36].

eBPF programs can use *helper functions* [4]. These helpers are functions implemented in the kernel to interact with the system. Using such helpers, eBPF programs can push and retrieve data to or from the kernel, and rely on mechanisms implemented in the kernel. For instance, the helpers can be used to print debugging messages, to interact with eBPF maps, or to manipulate network packets. Figure 2 shows all available bcc/BPF tracing and performance tools.

C. XDP

eXpress Data Path (XDP) [16] is a new programmable layer in the kernel network stack. It offers a run-time programmable packet processing inside the kernel instead of kernel-bypass. XDP provides access to functions implemented in network drivers and offers an API for fast packet processing across hardware from different vendors.

The XDP system offers several compelling advantages over DPDK and other kernel bypass solutions [16]. Specifically:

- XDP requires no changes to network configuration and management tools and retains the kernel security boundary.
- Ease in adding XDP execution hooks.
- Acceleration of critical performance paths when selecting the kernel network stack features such as the routing table and TCP stack.
- Transparency to applications running on the host, enabling the deployment of security rules such as DoS attacks.
- It can be dynamically re-programmed, i.e the features can be added/removed on the fly without interruption of network traffic.
- Less overhead with lower CPU usage and power saving implications.
- It provides a Linux friendly alternative to DPDK. It allows the user to custom and process packets that scale linearly with CPU cores.

XDP is actively used. Cloudflare integrated XDP into their DoS mitigation pipeline [7]. Suricata have XDP plugins [37]. Facebook released to use XDP as a high-performance layer 4 load balancer [17].

D. PCP/Vector

Vector [22] is a browser-based performance monitoring and analysis tool, developed by Netflix for massive-scale real-time system monitoring. It depends on the Performance CoPilot (PCP) [2] system, a standard way to collect and analyze different types of system metrics. As a data visualization tool, the two components are easily configurable and very lightweight.

III. Motivation

Our goal is to demonstrate the power of eBPF/XDP and replace the existing ROS command line tools with a monitoring system that can maintain metrics on every single topic and service within a ROS system while adding minimal overhead for the embedded systems. Our system has to be extensible as well, to allow users to maintain the flexibility that is inherent to ROS. We demonstrate the flexibility of our system by building two separate modules that target common use cases with ROS systems, security, and monitoring.

While the topic of security in ROS has been gaining more academic attention over the past few years [13] [23] [35] the

field is still relatively new. Currently, security for ROS1 is centered around hardening nodes and applying cryptography to communications, though there has been some research into applying security at the network layer. Our module builds on existing network layer security for ROS [27] but demonstrates the performance improvement available with the use of eBPF.

There are many options available for monitoring multiple ROS systems, with the most common being AWS metrics [5] and Overseer [29]. While both of these options do support monitoring capabilities for ROS, they both have performance comparable to the native command-line ROS tools. We demonstrate a module that performs better than the native monitoring tools, that can be scaled to similarly large numbers of robots. Additionally given that the ROS ecosystem is currently in a state of transition between ROS1 and ROS2, we designed our system to support both, in order to provide longevity and value to more users.

IV. Related work

Bihlmaier et al. [8], [9] proposed ARNI, a framework to monitor and introspect large ROS systems at run-time to find configuration errors and bottlenecks. Its main purpose is to collect and visualize information about the message flow inside the distributed network system. Additionally, it provides information about system resources such as CPU and memory for hosts and individual ROS nodes. ARNI allows visualizing using a dashboard. Additionally, it proposes countermeasures that can be taken to ensure continued functionality of the ROS network such as the detection of a violation of a known state.

Monajjemi et al. [21] proposed Drums, a lightweight distributed monitoring system resources, and a debugging tool for robot systems. System resources are PIDs or sockets of ROS nodes. Drums are used as a component for testing, debugging, and run-time quality-of-service monitoring. However, these network monitoring tools do not provide:

- deep network information such as net-flow data in a network system.
- new means of communication.

Rivera et al.[27] proposed ROSDefender, a network layer monitoring and security tool for ROS systems. It leveraged SDN components to provide packet-level filtering and analysis, as well as system-wide anomaly detection and correlation. While this tool provided both security and monitoring, it came with a heavy performance cost.

The security for ROS2 is a less established field, however, Kim et al[19] provided an exploratory look into the challenges and performance considerations for ROS2 systems with the secure DDS standard. They find that the current implementations of secure DDS add an overhead of around 100%, which is still usable for critical computing, however, they did caution that a more efficient implementation of the standard was possible.

When eBPF was fully implemented into the kernel, the security possibilities it afforded to researchers were quickly made apparent. Tian et al.[33] proposed LBM, a security framework for the Linux peripherals. They demonstrated the power of eBPF on the USB stack, the Bluetooth and the NFC peripherals stacks, and showed that rules-based enforcement was a notable improvement in the security of a Linux system. Similarly, Deepak et al. [11] demonstrated the viability for eBPF to replace the traditional iptables firewall application in Linux. Their solution still requires the manual port configurations that iptables do. Similar research was performed by Scholz et al.[32], which looked at the performance costs of an eBPF/XDP firewall vs the traditional iptables firewall. Finally, Deri et al.[12] was a recent advancement in the use of eBPF and XDP, for total system monitoring, leveraging the tables available to integrate information between various services from a system-level perspective.

Tran et al.[34] looked at the extensibility of TCP systems with eBPF. They demonstrated that inline packet processing for TCP was efficient and easy to implement with the use of eBPF. Given that ROS and ROS2 both leverage a similar set of TCP options, we found that the framework and hooks established in this paper were an effective starting out point for our efforts.

Our work builds on the previous work done to secure ROS systems in ROSDefender and extends it to support ROS2, while also building on the eBPF firewall components, providing the first eBPF firewall for systems that relies on layer 4.5 routing such as TCPROS and DDS. We vastly improve on the performance of the existing ROS monitoring tools while improving the overall security of ROS systems.

V. ROS monitoring tool

A. Design Goals and Overview

In this section, we describe the implementation of our fast ROS monitoring tool, namely *ROS-FM*. The tool was built to be able to provide monitoring at its core and an extensible interface for other functionality. We have implemented both a security module and a data visualization module. The core monitoring module is built to monitor the entire ROS system, leveraging the performance available with eBPF. The security module is built on XDP and provides rule-based filtering for ROS systems as well as protection from both network-level and application-specific attacks against ROS. Finally, the data visualization module is designed to export all of the monitoring and security information into Netflix's Vector[22] visualization for easy use.

B. Architecture Overview

ROS-FM is built as a central monitoring core and a collection of modules. Our system works by integrating the ROS core communication code with the kernel. The eBPF code operates in kernel space to filter all ROS middleware communications. The **core** acts as a translation tool between eBPF and the ROS framework, providing users with multiple different interfaces to manipulate ROS system data and traffic. To interact with and communicate with the core, we defined a simple rule-based Domain-Specific Language, discussed in Section V-C, as well as data hooks. We built two example modules to demonstrate the power of this construction, a **monitoring module** and a **security module**. A **visualization**

module is utilized to export data for visualization in PCP. Other modules are possible based on user input.

C. Domain-Specific Language

To facilitate user interactions with our system, we defined and implemented a domain-specific language that provides access to the underlying eBPF implementation. We leverage this DSL in both the monitoring and security modules. The language defines a set of simple rules that are automatically compiled into eBPF/XDP C by the core. This DSL is defined using the same primitives as ROS to allow for ease of application by ROS users.

Each rule in the *ROS-FM* system is written as a single line using the grammar outlined in Listing 1. The beginning of each line specifies the installed module the user wishes to apply the rule to. As a user installs additional modules, new rules can easily be designed by simply extending the grammar specifying the new module. Once specified, an expression is constructed using ROS native syntax to aid the bridge between the systems. The user can chain any number of expressions together to meet a required specificity. As an example, if a user wishes to constrain access to the 'map' topic, to a set group of nodes(A, B, C), then the rules would be:

```
Filter: TOPIC == "map" && NODE == "A";
Filter: TOPIC == "map" && NODE == "B";
Filter: TOPIC == "map" && NODE == "C";
```

The majority of the operands are straightforward, regarding topics, nodes, and messages in the same way as the ROS framework. However, the LeftOperand 'CUSTOM' is used for module-defined behavior.

Listing 1: Domain-Specific Language Grammar

```
1  Rule:
2      (Module ':' Expression ';')+
3  ;
4
5  Module:
6      'Filter' | 'Monitor' | 'Display'
7  ;
8
9  Expression:
10     SimpleExpression '&&' Expression
11     | SimpleExpression
12 ;
13
14 SimpleExpression:
15     LeftOperand op RightOperand
16 ;
17
18 LeftOperand:
19     'TOPIC'
20     | 'NODE'
21     | 'MESSAGE'
22     | 'FIELD(' message_field=STRING ')'
23     | '{CUSTOM}'
24 ;
25
26 op:
27     Equal
28     | NotEqual
29     | Contains
30     | ContainedIn
31     | Matches
32 ;
33
34 Equal: '==';
35 NotEqual: '!=';
36 Contains: 'CONTAINS';
37 ContainedIn: 'IN';
38 Matches: 'MATCHES';
39
40 RightOperand:
41     STRING
42     | '[' (STRING)+ ']'
43 ;
```

D. eBPF/XDP for ROS

The monitoring module operates on two different layers of the network stack taking advantage of the capabilities of XDP and eBPF. Much of the filtering is done at the XDP level, while the decision making is instead passed up to the eBPF socket layer and the main processing node. The program takes advantage of eBPF hash tables to extract data from the kernel. The first hash table is a mapping of a source port and destination port, with their respective IP addresses, with a rule for XDP filtering (Filtering Table). The second table is used by XDP to keep track of any metrics the user desires (Flow Metrics Table). With every new packet, the XDP program updates the relevant stream for these metrics.

Any new packets that are not in the Flow Metrics Table are passed up to the socket layer with a meta-data note. The socket layer also maintains two tables of its own. It maintains a translation table for all of the topics and services (Topic/Service Table) as well as a lookup table for all of the rules for the security module (ROS Rules Table). The socket layer has two core functions: it analyzes all the incoming packets that the XDP layer is unsure of to determine what the topic is, and it maintains a lookup table to track the location of every ROS component. Additionally, if the security module has been enabled, it takes advantage of the socket layer code to passively monitor all known node ports to ensure that the XMLRPC (Section II) exploits do not function. Essentially, the socket program uses its privileged location to ensure that nodes do not have the API called out of turn and that any API calls are from known sources. A visual representation of this system can be seen in Figure 3, stacked against a representation of the OSI model for comparison. The ROS Rules Table and the Flow Metrics Table are the interfaces for the security and visualization modules respectively.

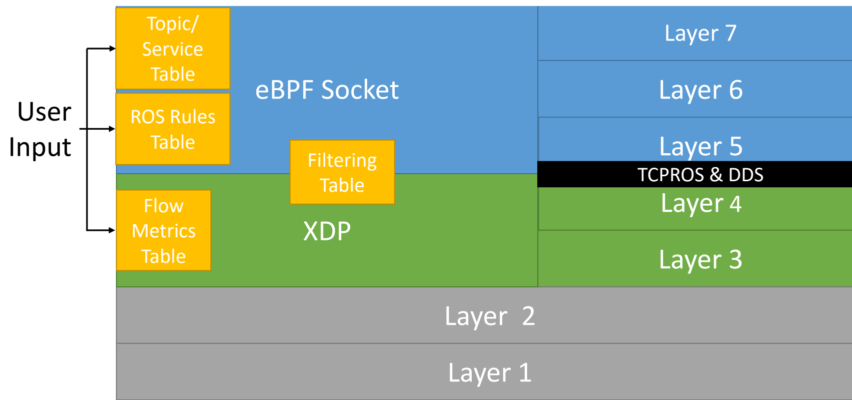


Fig. 3: Visual representation of monitoring system

The monitoring module provides the following custom commands:

- Log: Stores a copy of all matched packets to disk with the location specified by a user as a RightOperand
- Count: Maintains a running count of all packets that match the filter
- Average: Maintains a running average of all bytes per second that match the filter.
- Max: Stores the current largest packet that matches the filter.
- Min: Stores the current smallest packet that matches the filter.

ROS2 is built on top of the Data Distribution Service (DDS), which still operates on the normal Layer 1-4 network stack. This means that the XDP layer is identical between ROS1 and ROS2 while the socket layer only has to parse different packet meta-data. We implement both a TCPROS/UDPROS filter and a DDS filter for the socket filter, as two separate eBPF programs. While both of these programs can be loaded simultaneously, our filters cannot trace packets across the *ros1-bridge* node, meaning that such traffic is lost. This was considered acceptable as the *ros1-bridge* appears to be rarely used, as do hybrid systems in general.

While eBPF places substantive limitation of both the size and complexity of programs, we were able to design our system such that the higher-order processing was done purely in user-space, with the kernel space programs following the rules set in place in user-space. This allows for a more extensive development environment, though it does come with the limitation of a delay before new rules are fully implemented.

For the security module, we take advantage of the ability of both eBPF and XDP to drop packets faster than user-space programs. The security nodule extends the monitoring capabilities at both the socket and XDP layers to add application-specific content. This is most notable in the socket layer where the security module, even with no other rules, enforces the ROS assumption for communication (a node must first communicate with a master node before it attempts to establish a connection with another node). This cuts off a wide variety of XML-RPC attacks against ROS which tend to function by exploiting the API. This is a vital function ROS does not provide. As an additional feature, we provide the user the ability to restrict access to certain function calls so that an attacker cannot update command line parameters or call a service it is not supposed to under penalty of dropped packets.

The firewall is configured as a default-deny filter, for all topics, and nodes that it has a rule for, and default-allow for those without. The firewall provides the following custom commands as used in ROSDefender [27]:

- Copy: Transmits a copy of all packets to another destination
- Limit: Allows the packets to pass up to the limit then drops the rest

We utilize the functionality of XDP to allow for the enforcement of consistent origins for messages from nodes. Once a node has been identified on the network, any attempts to spoof that node from a different source address are detected and blocked. This functionality extends to anonymous nodes as we do not intend to restrict any ROS features.

E. Visualization Module

The visualization module is designed to give users a consistently updating sense of their robots' network traffic and state. It is designed to allow each robot to export every single metric that is available through the robot command-line interfaces automatically, with low overhead. This module exports data in the PCP format.

We take advantage of Netflix's Vector's ability to pull and plot data from a large number of sources to export chosen metrics from multiple robots to a convenient location. This allows users to give an intuitive sense for their robots and quickly respond to any issues or attacks.

F. Implementation

ROS-FM is implemented in Go using gobpf to access the eBPF and XDP bindings. Go was chosen due to its higher

performance [1]. Each of the components of the system is encapsulated in their threads, with shared data buffers to share information between them. We leveraged many of the efficient parallelization mechanisms in Go including the channels and the dynamic rewrite semaphores. The core monitoring program provides an API for the other subsystems to use to provide a consistent interface for any modules available.

The security module is implemented as an extension to the table translation interface between topics and services. Given a defined rule file, which is structured to be similar to a ROS launch file, a user can specify which nodes are allowed to communicate on which topics, call which services, and how much data nodes are allowed to use to protect against DDOS and other broadscale network attacks. The rules file for the security tool does not have to be static and can be updated regularly to take advantage of existing research in machine learning and anomaly detection for ROS[27].

The visualization module is implemented as a PDMA plug-in using the Go PDMA library. It exports to the *mmv* module for PDMA which is designed for custom user plug-ins. The *mmv* module uses shared memory to monitor processes on the system. From there, any data visualization tool that can process PCP can render the data. For the visualization in Vector, we built templates to render the most common ROS metrics: number of topics, packets per topic, bandwidth per topic, and number of nodes and subscribers on a topic over time.

VI. Experiments

We defined the following research questions:

- Q1. What is the overhead of using *ROS-FM* vs the state-of-the-art ROS tools?
- Q2. What is the break-even point between the ROS tools and *ROS-FM*?
- Q3. How effective is *ROS-FM* at preventing common ROS network attacks?

We answered this question by performing the experiments described in this section.

A. Experimental Setup

We implemented our monitoring system in Go, using the gobpf library to load the eBPF code. We developed the eBPF code in C and built it at run time within the code. All experiments were carried out on a 3.2GHz Intel Xeon E5 virtual machine, 8 GB RAM, running Ubuntu 18.04, running kernel 5.0.0. We chose to use ROS1 Melodic and ROS2 Crystal, using the release versions of each. All tests were carried out with the security and display modules enabled to create a 'worst case' measure of overhead.

The code is available for download on Github at https://github.com/seanrivera/ROS_FM.

We performed the following experiments and summarize the results:

- Compare the overhead induced by the rostopic CLI tools with the overhead induced by our eBPF tools. We test the overhead in terms of processing cycles introduced on a ROS system with 1, 5, 10, 20, 40, 60, and 80 nodes communicating on 1 topic.
- Repeat the overhead measures with more topics, invoking the CLI tools on every single topic.
- Compare the accuracy of the eBPF performance tools against the Amazon ROS performance tools.
- Evaluate the overhead on a ROS2 system.
- Using industry-standard security tools ROSPenTo and ROS2-SecTest to determine how effective the security plugin is.

B. Overhead analysis

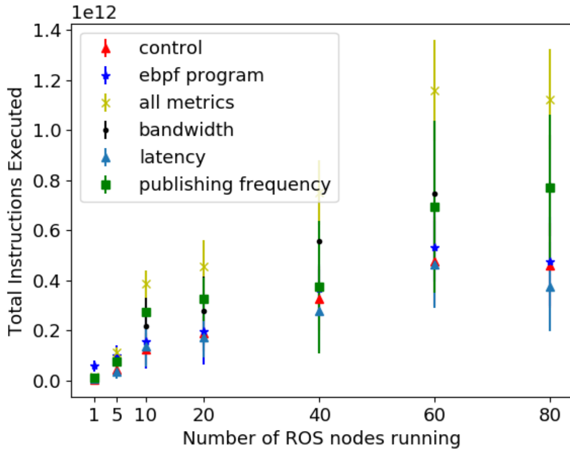
In analyzing ROS1 overhead, we found that our system has a processor overhead between 660% for a ROS system with a single publish/subscribe node pair and 1.6% for a system with 80 publish-subscribe nodes. It is important to note that these are the 'worst' and 'best' case scenarios respectively. In the average case (5-8 nodes), our system performs comparably to all ROS metric tools, while simultaneously providing additional security. However, if the user is only concerned with a single ROS metric, our tool provides the benefit of added security but has a moderately higher performance overhead. In cases with ten nodes or more, our tool far outperforms all others with an overhead of 23%, as compared to 73% for a single topic or 205% for all metrics.

The high-performance overhead, when compared to the single node system, is due to the overhead of the userspace program and the eBPF compiler. The singular ROSTopic metric's performance overhead is found to have an overhead between 34% and 52%. Once you include all of the ROStopic metrics that are provided by the eBPF program, the performance overhead of the CLI tools instead ranges from 105% and 140% depending on the number of nodes. Once there are more than 5 node pairs on the system, we find that the eBPF program outperforms the equivalent ROS tools needed to provide the same metrics. Additionally, we find that the eBPF program exceeds the performance of the monitoring tools completely after 10 nodes pairs have been added to the system. On the far end of the spectrum, our eBPF monitor outperforms the standard ROS tools by a factor of 25. We demonstrate these overheads in Figures 4a and 4b, with Figure 4a demonstrating how many instructions were used over 120 seconds of run-time and Figure 4b demonstrating the overhead of our tool and the command line tools as compared to the control 120 second run.

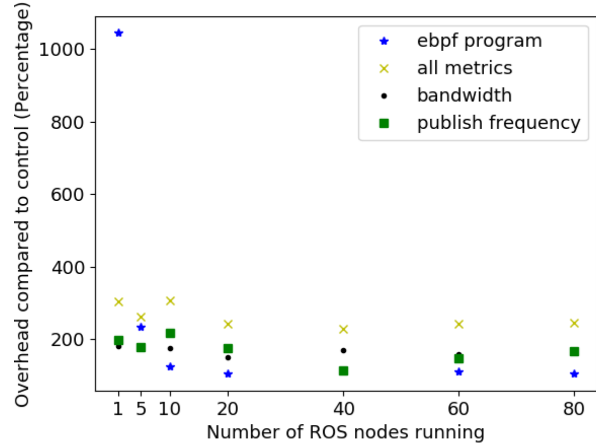
In ROS2, we find that the overhead is instead between 515% and 15% for the eBPF program and 80% to 110% for the ROS2 CLI tools. We found that the current ROS2 master and node paradigm had a higher startup cost but that was offset by the overhead of parsing the DDS packets. We analyzed ROS2 using the eProsima DDS provider.

C. Security Evaluation

In order to evaluate the security provided by our solution, we compared it to two existing security tools for ROS1 and ROS2, ROSPenTo[14] and Amazon's ROS2 Security test



(a) Overhead comparison of ROSCLI tools with ebpf program



(b) Plot of the overhead introduced by the the CLI tools vs the ebf system

Fig. 4: Analysis of ROS1 tools vs *ROS-FM*

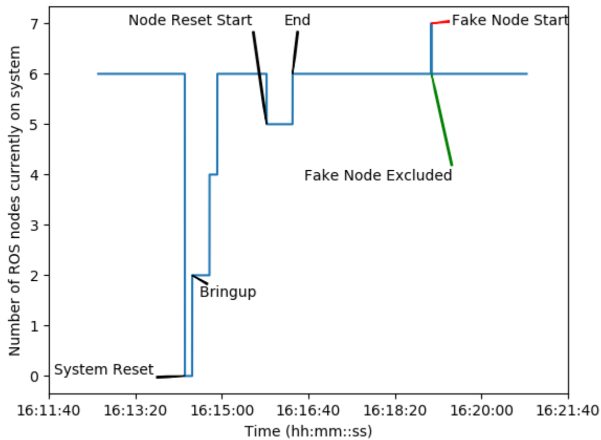


Fig. 5: Node tracking example

Attack	Result
Add publisher	Negated
Replace publisher list	Negated
Remove publisher	Negated
Isolate Service	Negated
Unsubscribe Node from Parameter	Limited
Update parameter	Limited

TABLE I: ROSPenTO

CPU Exhaustion	Successful
Network Exhaustion	Negated
Disk Exhaustion	Successful
Memory Exhaustion	Successful

TABLE II: AWS ROS2 SECTest Node

node[6]. These two systems are the current best standard test tools available to our knowledge, though, like all penetration tools, they are only a starting point for security comparison. We activated the two tools on a system secured with the XDP module and summarized our results in the Tables I and II. We found that we were able to completely negate any attacks that relied on out of order execution of exploits to the node API directly, such as those attacks that change the publisher or subscriber status. Additionally, we were easily able to protect services from isolation and calls by unauthorized nodes. While we found that attacks between nodes and the parameter server were still possible, they were rendered more difficult as they could only be launched from the same source as the valid node, and they could only affect the parameters the node has access to. The addition of PKI renders all attacks against the parameter server impossible. Unfortunately, our system is far less effective against the ROS2 SEC-Test attacks, as we can only negate the network resource exhaustion effects and provide no protection against the other resource exhaustion effects at present. A future extension to the security module could take advantage of eBPF's socket options in that area to limit nodes' access to computing resources.

We confirmed that the addition of Secure-ROS, SROS, and the secure DDS extension for eProsima do not heavily impact the performance of the eBPF monitoring tool. However, the socket layer analysis is more limited in cases where encryption is used. Once the packets are encrypted we can no longer enforce limitations on the XMLRPC calls, or any other rules that require knowledge of the packets.

D. Monitoring Evaluation

We demonstrate the appearance of the exported data for the number of topics active on a system during normal operation in Figure 5. The PCP plugin also exports the current number of packets sent on the topic, the current bandwidth used by a topic, and the average packet size per topic. All of these topic-

specific metrics are dynamically added and removed from the PDMA exporter at runtime.

E. Comparison and discussion

We found that the eBPF tool exceeded the performance of the traditional ROS CLI tools once the system reached a sufficient level of complexity, and the initial overhead of loading eBPF is proportionally small. We also found that we can monitor both ROS1 and ROS2 with similar overhead, though ROS2 carries a higher overall overhead. We demonstrated all of this while providing both security protection and metric visualization for users.

VII. Conclusion and future work

In this paper, we proposed a *ROS-FM* novel system to monitor robotic systems built on top of ROS. We demonstrate that this system provides better performance than the native ROS1 tools and still supports the same monitoring for ROS2. Furthermore, we build two modular extensions to the system, a security module, and a data visualization module that leverages Netflix's Vector tool to render the results for easier user experience. We evaluate the security module against standard security tools for both ROS1 and ROS2 and show the improvements gained from using our system.

While we believe that our system can be added to any singular robotic system, there are some improvements we would like to make for future research. Firstly, we would like to test our system with additional ROS2's DDS implementations to evaluate what other metrics are valuable to track. Furthermore, we would like to expand our system to cover multiple separate robots, sharing their hash tables and rules to track distributed robotic systems. We would also like to build other modules to address other needs in the robotics community with eBPF.

Our main contributions are:

- Monitoring: An eBPF monitoring framework for robotic systems with very low overhead.
- Security: A network layer security add-on to extend the core ROS security model.
- Usability: A PCP plugin to export and easily visualize robotic statistics.
- Reproducibility: Our code and data are publicly available.

VIII. Acknowledgement

This work is partly supported by the project CONCORDIA that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 830927.

References

[1] Go versus python 3 fastest programs. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go-python3.html. [Online; accessed 30-August-2019].

[2] Performance co-pilot. https://pcp.io/. [Online; accessed 30-August-2019].

[3] Bpf and xdp reference guide. https://docs.cilium.io/en/v1.4/bpf/, 2019. [Online; accessed 30-August-2019].

[4] Bpf helpers. https://qmonnet.github.io/bpf-helpers/out/bpf-helpers.html, 2019. [Online; accessed 30-August-2019].

[5] AWS-Robotics. Health metric collector. https://github.com/aws-robotics/health-metrics-collector-ros1, 2019.

[6] AWS-Robotics. Ros2-sectest. https://github.com/aws-robotics/ROS2-SecTest, 2019.

[7] G. Bertin. Xdp in practice: integrating xdp into our ddos mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, vol. 2, 2013.

[8] A. Bihlmaier, M. Hadlich, and H. Wörn. *Advanced ROS Network Introspection (ARNI)*, pages 651–670. Springer International Publishing, Cham, 2016.

[9] A. Bihlmaier and H. Wörn. Increasing ros reliability and safety through advanced introspection capabilities. In E. Plödereder, L. Grunske, E. Schneider, and D. Ull, editors, *Informatik 2014*, pages 1319–1326, Bonn, 2014. Gesellschaft für Informatik e.V.

[10] B. Daniel. On getting tc classifier fully programmable with cls bpf. In *Proceedings of netdev*. 2016.

[11] A. Deepak, R. Huang, and P. Mehra. ebpf/xdp based firewall and packet filtering. In *Linux Plumbers Conference*, 2018.

[12] L. Deri, S. Sabella, and S. Mainardi. Combining system visibility and security using ebpf. In *ITASEC*, 2019.

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

[14] B. Dieber, R. White, S. Taurer, B. Breiling, G. Caiazza, H. Christensen, and A. Cortesi. Penetration testing ros. In *Robot Operating System (ROS)*, pages 183–225. Springer, 2020.

[15] P. Estefo, J. Simmonds, R. Robbes, and J. Fabry. The robot operating system: Package reuse and community dynamics. *Journal of Systems and Software*, 151:226–242, 2019.

[16] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 54–66, 2018.

[17] F. Incubator. A high performance layer 4 load balancer. https://github.com/facebookincubator/katran, 2019. [Online; accessed 30-August-2019].

[18] IOVISOR. Bpf compiler collection (bcc). https://github.com/iovisor/bcc, 2019.

[19] J. Kim, J. M. Smereka, C. Cheung, S. Nepal, and M. Grobler. Security and performance considerations in ros 2: A balancing act. *arXiv preprint arXiv:1809.09566*, 2018.

[20] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[21] V. Monajjemi, J. Wawerla, and R. Vaughan. Drums: A middleware-aware distributed robot monitoring system. In *2014 Canadian Conference on Computer and Robot Vision*, pages 211–218, 2014.

[22] Netflix. Vector. https://github.com/Netflix/vector, 2019.

[23] D. Portugal, M. A. Santos, S. Pereira, and M. S. Couceiro. 20 on the security of robotic applications using ros. 2017.

[24] R. Project. Ros technical overview. http://wiki.ros.org/ROS/TechnicalOverview. [Online; accessed 30-August-2019].

[25] R. Project. Tcpros. http://wiki.ros.org/ROS/TCPROS. Accessed: 2019-08-30.

[26] R. Project. Udpros. http://wiki.ros.org/ROS/UDPROS. Accessed: 2019-08-30.

[27] S. Rivera, S. Lagraa, C. Nita-Rotaru, S. Becker, et al. Ros-defender: Sdn-based security policy enforcement for robotic applications. In *IEEE Workshop on the Internet of Safe Things, Co-located with IEEE Security and Privacy 2019*, 2019.

[28] robotics business review. Global spending on robots, drones to top $115b in 2019, says idc). https://www.roboticsbusinessreview.com/manufacturing/global-spending-on-robots-drones-to-top-115b-in-2019-says-idc/. [Online; accessed 30-August-2019].

[29] F. Roman, A. M. Amory, and R. Maidana. Overseer: A multi robot monitoring infrastructure. In *ICINCO (1)*, pages 151–158, 2018.

[30] A. Santos, A. Cunha, and N. Macedo. Static-time extraction and analysis of the ros computation graph. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 62–69. IEEE, 2019.

[31] A. Santos, A. Cunha, N. Macedo, R. Arrais, and F. N. Dos Santos. Mining the usage patterns of ros primitives. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3855–3860. IEEE, 2017.

[32] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle. Performance implications of packet filtering with linux ebpf. In *2018 30th International Teletraffic Congress (ITC 30)*, volume 1, pages 209–217. IEEE, 2018.

[33] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, P. C. Johnson, and K. R. Butler. Lbm: A security framework for peripherals within the linux kernel. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (IEEE S&P)*. 2019.

[34] V.-H. Tran and O. Bonaventure. Beyond socket options: making the linux tcp stack truly extensible. *arXiv preprint arXiv:1901.01863*, 2019.

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

[36] M. Xhonneux, F. Duchene, and O. Bonaventure. Leveraging ebpf for programmable network functions with ipv6 segment routing. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 67–72, 2018.

[37] L. Éri and M. Peter. White paper: Introduction to ebpf and xdp support in suricata. Technical report, 2019.