

ESROS2: Application Sandboxes for Secure ROS2

Dev Tejas Gandhi
Indian Institute of Science
Bangalore, India
devtejas@iisc.ac.in

Vinod Ganapathy
Indian Institute of Science
Bangalore, India
vg@iisc.ac.in

Abstract

ROS2 is a popular middleware for several robotics platforms. It allows rapid prototyping of distributed robotics applications via a convenient publish/subscribe-based abstraction for application communication. Secure ROS2 (SROS2) is a set of patches to ROS2 that provides several basic security primitives to secure application communication, such as encrypted communication and limiting applications to only publish/subscribe to topics that are declared in a manifest.

Unfortunately, SROS2 is limited in its ability to confine application communication. Applications can freely communicate with each other by bypassing the SROS2 API and directly making system calls via the underlying operating system. This paper introduces ESROS2, which provides application sandboxing in SROS2. ESROS2 confines applications by limiting the set of system calls that they can issue at runtime. When an application issues a system call, ESROS2 determines whether the context in which the system call was issued is safe, and only then permits the system call to be issued. We describe the design of ESROS2 and its implementation using extended Berkeley Packet Filters (eBPF) and commodity intra-process memory isolation hardware (Intel MPK). We show that ESROS2 works on off-the-shelf application binaries, and effectively enforces application sandboxing while only imposing a modest runtime performance overhead.

CCS Concepts: • Security and privacy → Software security engineering.

Keywords: ROS2, application sandboxing, extended Berkeley packet filters, intra-process isolation

ACM Reference Format:

Dev Tejas Gandhi and Vinod Ganapathy. 2026. ESROS2: Application Sandboxes for Secure ROS2. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026)*, November 15–18, 2026, Hong Kong, Hong Kong. ACM, New York, NY, USA, 17 pages. <https://doi.org/10.1145/3828523.3857133>



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

SIGOPS ATC 2026, Hong Kong, Hong Kong

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11

<https://doi.org/10.1145/3828523.3857133>

1 Introduction

ROS2—the Robot Operating System (version 2) [74, 75]—is a popular middleware for the development of distributed robotics platforms [3, 29, 70]. It has achieved widespread adoption, as evidenced by approximately 897.5 million package downloads in 2025 and its use by more than 1,579 companies worldwide [81]. ROS2 eases the development of distributed robotics applications via a publish/subscribe paradigm that allows applications to conveniently communicate with each other. A data-producing application such as the front-end to a sensor (e.g., LIDAR or GPS) publishes sensor-related data labeled with a *topic name*. A data consumer that wishes to use this data simply subscribes to this topic. ROS2 does the heavy-lifting of setting up the underlying communication channel between the producer and consumer—even if they run on different hardware platforms. Applications are thus shielded from the complexities of dealing directly with hardware.

ROS2 does not natively sandbox applications or secure their communication. Data is transmitted in clear text, and topic-based access control is advisory, enabling attacks such as data spoofing and publisher/subscriber impersonation. *Secure ROS2 (or SROS2)* [18, 46, 83, 91, 93, 94] rectifies this situation by providing several security features. It ensures that the publish/subscribe behavior of an application is constrained by the list of topics to which it publishes/subscribes, as declared in a manifest that is digitally signed by the application developer. At runtime, SROS2 ensures that an application cannot publish or subscribe to topics outside this list. SROS2 also sets up TLS connections for inter-application communication. Together, these measures prevent data spoofing and ensure data confidentiality, integrity, and sender authentication.

Unfortunately, SROS2’s mechanisms are insufficient. This is because SROS2 does not prevent applications from bypassing the publish/subscribe mechanism. This loophole voids the intent of SROS2’s sandboxing. For example, a publisher can easily communicate with a subscriber by setting up socket communication or shared memory, which fall outside the purview of the SROS2 publish/subscribe API. Malicious applications or exploits against vulnerable applications can misuse this ability. Numerous such vulnerabilities have been reported either in SROS2 applications, or libraries (such as OpenCV and zlib) that are used extensively in SROS2 applications [13, 51–65]. Exploits of these vulnerabilities can allow an attacker to execute arbitrary code with the privileges

We are grateful to our shepherd, Indranil Gupta, and the reviewers for their feedback. We thank the K-Tech CoE on cyber security for financial support.

of the vulnerable application. This motivates the need for a security mechanism to tightly sandbox applications so that their communication is governed by SROS2’s security policies.

This paper introduces **ESROS2**, which achieves this goal by observing that any application that wishes to communicate outside SROS2’s publish/subscribe abstractions must necessarily do so by invoking the underlying operating system’s communication abstractions, *i.e.*, by invoking system calls. ESROS2 restricts applications from issuing system calls directly, thereby forcing them to use SROS2’s publish/subscribe API. This API enforces SROS2’s security policies, ensuring communication between applications only when the published and subscribed topics match.

Completely restricting an application from issuing system calls breaks functionality due to the way in which SROS2’s software stack is structured. SROS2 consists of a layered stack of libraries that are dynamically linked to the application. These libraries themselves issue system calls to implement the core functionality of SROS2’s communication abstractions. Simply blocking *all* system calls issued by an application will thus break functionality. ESROS2 thus adopts a policy that allows system calls to be issued by SROS2 libraries but not by the application code. When an application issues a system call, ESROS2 determines the context in which the system call is issued. It then enforces a simple invariant called **ESROS2-invariant**: System calls can be issued from the trusted computing base of ESROS2, but cannot be issued directly from the application’s business logic or other supporting libraries, unless they are explicitly allowed by a security analyst. The latter case may be needed in legitimate situations in which the application invokes utilities from standard libraries such as `libc` or `libstdc++`, which in turn invoke system calls. **ESROS2-SCgate** is a mechanism to allow the application to invoke system calls from standard C/C++ libraries, provided such calls are explicitly permitted by an application-specific security policy.

ESROS2 is tailored to SROS2 and is better suited to its security requirements than existing techniques such as `seccomp` or container-based system call sandboxes [15, 22, 27, 30–32, 66, 68]. Prior system call sandboxing techniques have largely provided static policies or are unaware of the (SROS2-specific) context in which a system call is issued. For example, a socket-related system call should be allowed when it is issued for legitimate ROS2 communication, but should be blocked when it is invoked by compromised application code for malicious purposes. An approach that uses an allow-list of system calls, or even one that adapts this list but is unaware of application context when a system call is issued would be unable to provide adequate defense. Additionally, prior approaches fail to efficiently handle multi-threaded applications and callback-based execution, both of which are features that SROS2 applications use extensively [23, 24, 26, 33, 42, 87, 89]. ESROS2 also does not require application source-code modifications.

SROS2 middleware and applications can execute atop a variety of underlying platforms. Our ESROS2 prototype is implemented for SROS2 applications running atop a Linux-based system. In particular, ESROS2 is implemented using the Extended Berkeley Packet Filters (eBPF) [21, 25, 80] framework. eBPF allows safe in-kernel execution of security policies and mechanisms specified in user-space. ESROS2’s mechanisms are implemented using eBPF, allowing easy deployment without requiring kernel-level code modifications. ESROS2 also leverages commodity intra-process memory isolation hardware—Intel Memory Protection Keys (MPK) [37] in our prototype—to safely store the application’s security context, which the eBPF module consults when deciding whether to permit a system call.

To summarize, the main contributions of ESROS2 are:

- Identifying a key shortcoming in SROS2 that allows applications to violate the intent of the publish/subscribe sandbox and communicate via system calls (Section 2);
- ESROS2-invariant, a powerful invariant that constrains the runtime system call behavior of SROS2 applications. It evolved based on our analysis of standard C library usage by SROS2 applications, which allows practical deployment with only a handful of rules in ESROS2-SCgate (Section 3);
- And an MPK and eBPF-based low-overhead implementation of ESROS2-invariant that applies to off-the-shelf ROS2 application binaries, with support for multi-threading, asynchronous and dynamically-invoked operations (Section 4).

We evaluated ESROS2 via case studies using three real-world robotics application suites—a robotic arm, night-vision image processing, and autonomous robot navigation—to understand the effectiveness of ESROS2-invariant and the additional effort required by the security analyst to write policies for ESROS2-SCgate. We also measured runtime latency, startup costs and memory overheads of ESROS2 by running benchmarks from the iRobot [38] suite (Sections 6 and 7).

2 ROS2, SROS2 and Motivation for ESROS2

ROS2 is a set of layered libraries as shown in Figure 1. Applications are written in C++, Java, or Python using the API presented by the `rc1cpp`, `rc1java` or `rc1py` libraries. These libraries provide the language bindings to ROS2’s main client library `rc1`, which implements abstractions such as ROS2 nodes and topic names. ROS2 allows applications to declare a set of topics to which they can publish or subscribe. For example, a `GPSSensor` application could read the GPS coordinates from the hardware sensor and publish these coordinates labeled using the topic `GPSCoords` (using the APIs presented by `rc1cpp/rc1java/rc1py` to publish messages). A downstream `Navigator` application simply subscribes to the topic `GPSCoords` to receive data published by `GPSSensor`. Each application can publish or subscribe to multiple topics. For example, `GPSSensor` and `Navigator` can also subscribe to a topic called

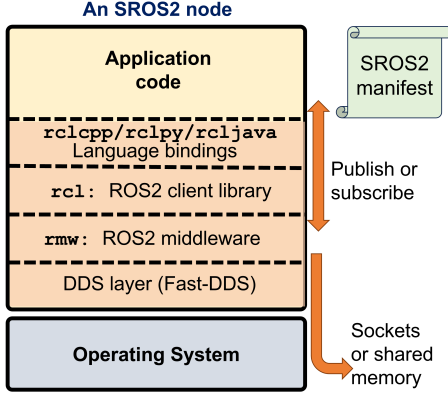


Figure 1. ROS2 supporting libraries and application communication—SROS2 is built as a set of patches to these libraries. For this paper, we are only interested in ROS2 enhanced with SROS2 patches. We therefore interchangeably refer to them as ROS2 or SROS2 libraries. In this paper, we will also refer to the collection of `rclcpp/rclpy/rcljava`, `rcl`, `rmw`, and the DDS libraries (*i.e.*, the user-space abstraction layers below the application that together implement ROS2), which form ESROS2’s trusted computing base as **ESROS2-TCB**.

SystemClock to which the system clock publishes time. The remaining layers such as the ROS2 middleware layer `rmw` and Data Distribution Service (DDS) [14, 35, 78] facilitate communication between publishers and subscribers via UDP sockets or shared memory.

By itself, ROS2 does not offer any security guarantees. Applications communicate using cleartext messages, which allows attackers the ability to implement a variety of attacks, including snooping on messages, spoofing the identity of the sender/receiver, publishing messages under arbitrary topics, and dropping/modifying published messages [16, 17, 19, 72]. SROS2 enhances the security of ROS2 by:

- (1) *Using end-to-end encryption:* SROS2 ensures that all publish and subscribe communication between applications happens over secure sockets (TLS), thereby ensuring confidentiality and integrity of data communication.
- (2) *Assigning identities to applications:* Each application on SROS2 must have an identity, represented by its X.509 certificate signed by a trusted third-party. This ensures that the identity of the applications on each end of the communication channel can be established.
- (3) *Restricting publish/subscribe using topics:* Application developers are expected to provide an accompanying manifest that declares the set of topics to which the application publishes and subscribes, respectively. Figure 2 presents a snippet from an example manifest. The manifest is cryptographically tied to the application using its X.509 credentials, thereby ensuring that any malicious modifications to the manifest that violate the intent of the application developer can be detected during certificate validation.

SROS2 constrains the runtime behavior of the application to only publish or subscribe to messages labeled with the

```
<permissions>
  <grant name="/GPSSensor">
    <subject_name>CN=/GPSSensor</subject_name>
    <allow_rule>
      <publish>
        <topics>
          <topic> GPSCoords </topic>
        </topics>
      </publish>
      <subscribe>
        <topics>
          <topic> SystemClock </topic>
        </topics>
      </subscribe>
    </allow_rule>
    <default> DENY </default>
  </grant> ...
</permissions>
```

Figure 2. Example of an application manifest in SROS2.

topics pre-declared in the manifest. This feature tightly sandboxes the application’s ability to publish or subscribe data and limits the set of applications with which it can communicate. In turn, this feature allows the implementation of other security primitives such as mandatory access control that allow a security analyst to reason about the flow of information in the entire system. For example, ROS2/AppArmor [5] and Privaros [7] build on SROS2 to implement mandatory access control policies that limit the set of downstream applications that can consume data produced by a data publisher.

Unfortunately, SROS2’s mechanisms are inadequate because applications may resort to directly using the primitives of the underlying system (via system calls) to communicate with each other. For example, the GPSSensor application (discussed earlier) publishes messages under the topic `GPSCoords`. However, GPSSensor can communicate its output to other applications (that may not necessarily subscribe to `GPSCoords`) via network sockets, the file system, or any of the other plethora of communication abstractions provided by the OS. SROS2 will be unable to mediate this communication via its publish/subscribe API. Even if an application is not overtly malicious in intent, it may contain vulnerabilities that an attacker can exploit to run arbitrary code with the application’s privileges. Indeed, there are several such CVEs reporting memory overflow and arbitrary code-execution in SROS2 software as well as libraries such as OpenCV and zlib that are extensively used by SROS2 software [13, 51–65].

2.1 Challenges and Goals of ESROS2

ESROS2 aims to strengthen SROS2 by restricting how applications invoke system calls, thereby ensuring that communication occurs through the SROS2-mediated publish/subscribe mechanism. This approach must address two challenges:

- *System calls from ROS2 libraries.* ROS2 is implemented as a stack of dynamically linked libraries (Figure 1), including `rclcpp`, `rcl`, `rmw`, and DDS. These libraries necessarily access operating-system resources, such as sockets and shared memory, and must be permitted to issue system calls.

- *System calls from standard C/C++ libraries.* ROS2 applications routinely invoke `libc` primitives. (We use “`libc` primitives” as a catch-all phrase that includes utilities from all C and C++ system libraries). While many of these utilities are benign (*e.g.*, `printf` and `malloc`), others can directly access system resources through file and network I/O.

To address these challenges, ESROS2 enforces context-aware system-call mediation by examining the execution context of each system call and determining whether it should be permitted. System calls originating from trusted ROS2 libraries are allowed by default, while those issued by application code are denied unless explicitly mediated through ESROS2-SCgate. By channeling communication through SROS2’s declarative publish/subscribe interface, ESROS2 strengthens information-flow control in SROS2.

2.2 Threat Model and TCB

ESROS2 assumes that the application is untrusted. The application is capable of issuing system calls—these could take the form of raw system calls (*e.g.*, via inline assembly instructions such as `syscall`) or by invoking `libc` primitives (or other libraries) that the application links against. However, we do assume that code injection attacks are precluded via data-execution prevention. This feature is available in most modern hardware, and eliminates code injection without binary modification or runtime overhead. We assume that the application could contain vulnerabilities that could lead to arbitrary code execution, *e.g.*, via code-reuse exploits [73].

Recall that SROS2 is implemented using the libraries of the ROS2 stack (those shown in Figure 1), and ESROS2’s goal is to enhance SROS2. Thus, in ESROS2 we consider those libraries as part of ESROS2’s trusted computing base, and collectively refer to them as **ESROS2-TCB**. The underlying operating system is also part of ESROS2’s TCB, as ESROS2’s core enforcement mechanism (implemented in eBPF) executes within the kernel. Furthermore, ESROS2-TCB includes ESROS2-SCgate, a set of wrappers that allows applications to declare additional system calls that they may need through `libc`. Finally, the system libraries on which the ROS2 stack depends, like `libc`, `libstdc++`, and `libm`, are also considered part of the TCB, as ESROS2-TCB relies on the functionality provided by these libraries.

ESROS2’s approach raises the bar for an attacker by restricting the application’s system call footprint to sequences of system calls that can be issued by ESROS2-TCB, and those declaratively permitted by ESROS2-SCgate. Out of scope for ESROS2 are attacks that accomplish their goal without a system call footprint, *e.g.*, by modifying data values in the application, or those that operate within the system call footprint permitted by ESROS2. Some of these attacks can be prevented by enhancing the application with control-flow integrity [1] or data-flow integrity [11], but ESROS2 does not assume that the application incorporates these defenses. Side-

or covert-channel attacks that can be used to leak information between SROS2 applications are also out of scope.

3 Sandboxing with ESROS2-invariant

ESROS2 sandboxes SROS2 applications by restricting the system calls that they can issue at runtime. Whether a system call is permitted depends on the *context* in which it is issued. Prior work has extensively explored context-sensitive system-call sandboxing [15, 23, 24, 26, 27, 30–34, 42, 66, 68, 89], typically by constructing models of application behavior from dynamic or static analysis [23, 24, 26, 33, 42, 89]. These models characterize legal system-call sequences, often using formal representations such as context-free languages [23, 33], and are considered precise when they closely match the application’s actual runtime behavior.

We chose not to adopt context-sensitive system-call sandboxing because of the need for precise static analysis to construct accurate system-call models, a challenge even for statically linked binaries [23, 33]. This problem is further complicated in SROS2 applications due to C++, virtual dispatch, dynamic linking, and multithreading, while efficient runtime enforcement remains costly in both runtime and memory overheads. An alternative is to use `seccomp` profiles [22], which are widely deployed in Linux and Docker containers and can be automatically generated for applications [10]. However, `seccomp` policies are not context-sensitive by default and rely on coarse-grained system-call filtering. In contrast, ESROS2 considers the origin of each system call—whether it originates from the application, ESROS2-TCB, `libc`, or another library—before permitting it.

3.1 ESROS2’s Invariant-Based Approach

Because ESROS2 aims to enforce adherence to SROS2’s publish/subscribe confinement policies, it must generally disallow system calls that are issued directly by the application. Of course, system calls are essential to interact with the underlying hardware, and the libraries in ESROS2-TCB themselves use system calls extensively. Blocking these system calls will break application functionality, so such system calls issued by ESROS2-TCB must be allowed. However, raw system calls issued by the application, or those that bypass ESROS2-TCB must be disallowed (unless explicitly allowed via ESROS2-SCgate policies). Thus, ESROS2 aims to enforce an invariant, called *ESROS2-invariant* (Figure 3), expressed using the runtime stack configuration of the application.

To illustrate ESROS2-invariant, consider the example stack configurations illustrated in Figure 3. Figure 3(a) and Figure 3(b) show scenarios in which an application attempts to communicate using ROS2’s abstractions, which leads to stack frames from ESROS2-TCB to be placed above those of the function’s own stack frames. In turn, ESROS2-TCB may either invoke a raw system call, as in Figure 3(a), or invoke a system call via `libc` primitives, as shown in Figure 3(b). In

ESROS2-invariant: When a system call is issued, the user-space stack of activation records must contain stack frames of ESROS2-TCB functions between the top of the stack and the stack frame of the most recently invoked entry point of the application’s logic.

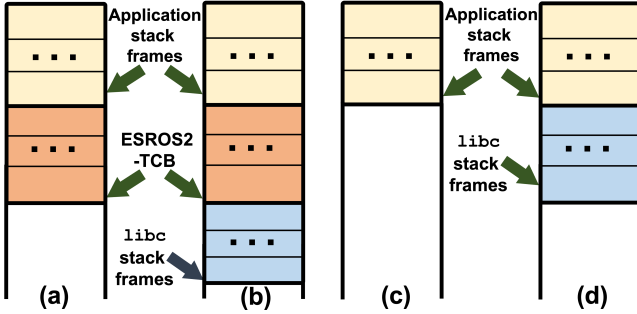


Figure 3. Examples of runtime stack configurations when control enters the kernel upon a system call. Following standard convention, stacks are assumed to grow downwards, and the frame at the bottom of each picture is the stack top. Scenarios (a) and (b) represent situations in which ESROS2-invariant is satisfied. Scenarios (c) and (d) represent violations of ESROS2-invariant.

both these scenarios, ESROS2-invariant is satisfied, and the system calls are allowed to proceed.

SROS2 applications make extensive use of callbacks [39]. Callbacks are functions that are registered with SROS2, which SROS2 then invokes when the system encounters an event of interest. ESROS2-invariant works robustly even in the presence of callbacks. When a callback is invoked, that function’s stack frame becomes the most recently-invoked entry point of the application’s logic. For ESROS2-invariant to hold, system calls from the callback must be via ESROS2-TCB.

The scenario in Figure 3(c) illustrates a case in which the application issues a raw system call, *e.g.*, using the `syscall` instruction (in x86-64) in inline assembly. This violates ESROS2-invariant, and represents a situation in which the application may bypass SROS2’s access control mechanism, and is therefore disallowed completely in ESROS2. Figure 3(d) illustrates a scenario in which the application invokes a `libc` primitive, which in turn invokes a system call. As such, this situation represents a violation of ESROS2-invariant. However, it also represents the all-too-common scenario in which an application avails of common `libc` utilities, the execution of which would result in several system calls. Disallowing the use of such utilities (because they violate ESROS2-invariant) breaks application functionality. On the flipside, the applications can use other `libc` primitives such as `fopen`, `socket`, or `write` to violate the SROS2 publish/subscribe confinement policies. Thus, a more nuanced approach is required to handle `libc` primitives when enforcing ESROS2-invariant.

3.2 `libc` and the SROS2 Application Ecosystem

SROS2 applications may invoke `libc` utilities to accomplish common tasks, which can violate ESROS2-invariant if the

Category	# Utilities
Process Control	272
Device Management	179
File Management	331
Protection	117
Communication	369
Information Maintenance	356
Unclassified	498

Figure 4. Triage results of utilities from `libc`. Terminology of categories is from a standard OS textbook [82].

`libc` utility invokes a system call, as in Figure 3(d). How do SROS2 applications commonly use `libc` primitives? How can ESROS2 support such use-cases while still respecting the intent of SROS2’s publish/subscribe access-control policies?

To answer these questions, we conducted a two-part study. Section 3.2.1 describes the first part of the study, in which we triaged `libc` primitives into various categories to understand which categories could potentially violate the intent of the SROS2 access-control policies. A SROS2 application that invokes `libc` primitives from such categories must explicitly declare its intent to do so, via a policy. Writing such policies requires effort. How much such effort is typically required for real-world SROS2 applications? Section 3.2.2 describes the second part of our study that answers this question by understanding typical uses of `libc` primitives in a suite of SROS2 applications. In Section 3.2.3, we describe how ESROS2-SCgate checks these policies at runtime.

3.2.1 Triaging `libc` Primitives. We analyzed `libc` (GNU C Library 2.35), which exposes 1,425 primitives, and categorized them into high-level OS functionality classes adapted from a standard OS textbook [82] (Figure 4). The goal was to identify `libc` functions that are potentially harmful when invoked by SROS2 applications. This manual classification is scalable because SROS2 applications and their dependent libraries invoke system calls exclusively through `libc` wrappers, making `libc` the sole system-call entry point during benign execution. We validated this assumption by analyzing over 500 SROS2 binaries and found no raw system-call instructions. We further confirmed that commonly used libraries, including `libstdc++` and `libm`, also rely on `libc` wrappers, indicating that intercepting `libc` is sufficient to mediate system-call invocations across the SROS2 stack.

Figure 4 presents the results of our triage. A single utility may belong to multiple categories, reflecting that it serves more than one purpose, *e.g.*, `mmap`, which is used both for memory management and memory-mapped file I/O. A utility is labeled Unclassified if it does not fall into any of the other categories or if it does not invoke any system calls (*e.g.*, string utilities such as `strlen` and `strcpy`). In total, we found that 721 utilities in `libc` belong to at least one of the first five categories in Figure 4. Utilities in these categories can generally be used to exfiltrate information by bypassing the SROS2 publish/subscribe interface, and their use by a

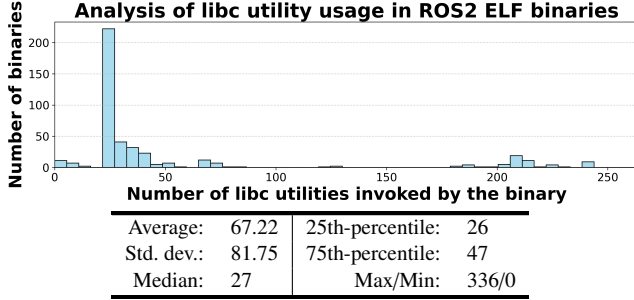


Figure 5. Analysis results showing the distribution of the number of libc utilities (from the first five categories of Figure 4) invoked by our suite of 456 ROS2 binaries.

ROS2 application must therefore be mediated by a security policy. Note that the above triage only serves as an aid to simplify categorization. We rely on a security expert to determine when security policies must be applied, via ESROS2-SCgate. For example, the use of `execve` and `dlopen`, which both fall under the Process Control category, must be mediated by a security policy because they can be used by an application to bypass SROS2. However, utilities in the same category such as `malloc` or `realloc` are more benign in intent and their use generally does not call for additional security checks.

3.2.2 How do SROS2 Applications Utilize libc? With the libc triage results in hand, we analyzed how real-world SROS2 applications invoke libc utilities, particularly those in the first five categories of Figure 4. We cloned all ROS-Humble packages available through the Ubuntu package repository [76]. Since SROS2 packages may contain both native executables and Python scripts, we restricted our analysis to ELF binaries, yielding 456 binaries. For each binary, we resolved all shared-library dependencies and analyzed the binary together with its dependent libraries, excluding those belonging to ESROS2-TCB. Starting from the application’s entry point (*e.g.*, `main`), our analysis recursively identified all reachable libc functions by following both direct calls and dynamically resolved calls through the global-offset (GOT)/procedure linkage table (PLT). During this traversal, any call path that entered an ESROS2-TCB function was terminated and not analyzed further. Thus every reachable libc invocation reported by our analysis originates entirely from application code or untrusted libraries. Since calls originating from ESROS2-TCB were excluded, every reachable libc invocation identified by our analysis corresponds to a configuration similar to Figure 3(d) and violates ESROS2-invariant.

Figure 5 presents the results of our analysis. The *x*-axis represents the number of libc utilities invoked by a binary that fall into one of the first five categories from Figure 4, while the *y*-axis shows the number of ROS2 binaries that fall into that bucket. The peak in the chart shows, for instance, that there are 222 binaries that each directly invoke a total of between 22 to 26 libc utilities. One application binary contained calls to 336 distinct libc utilities—the maximum

```
typedef int (*FdPolFn)(int fd, const char *mode);
FILE *fopen_wrapper(const char *path, const char *mode) {
    mpk_entry_gate();
    int fd = open(path, flags);
    if (fd < 0) {
        mpk_exit_gate();
        return NULL;
    }
    FdPolFn policy = get_policy_function();
    if (!policy || !policy(fd, mode)) {
        close(fd);
        mpk_exit_gate();
        return NULL;
    }
    FILE *fp = fdopen(fd, mode);
    mpk_exit_gate();
    return fp;
}
```

Figure 6. Example of ESROS2-SCgate wrapper for libc. Section 4 explains the role of `mpk_entry_gate` and `mpk_exit_gate`.

observed. Although there are a few applications that do invoke several hundred libc utilities, the median number is 27. The key significance of this result is that it indicates a conservative upper-bound on the developer/security analyst effort involved in creating security policies for ESROS2-SCgate. In our case studies, we experimentally quantified this effort and showed that it is relatively modest for most applications.

3.2.3 System Calls with ESROS2-SCgate. A SROS2 application that directly invokes a libc utility will find itself violating ESROS2-invariant if that utility invokes a system call. However, as we have just seen, ROS2 apps do use libc utilities. ESROS2-SCgate is ESROS2’s mechanism to allow applications to declare their use of libc utilities.

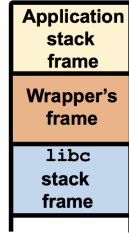
ESROS2-SCgate is a set of wrappers for libc utilities. Wrappers follow a standard template as described below. ESROS2 provides a wrapper generator to automatically generate wrapper code for a given libc API. We generate such wrappers for the libc utilities in the first five categories from Figure 4. The wrapper of a libc utility has a function prototype matching that of the utility. The body of the wrapper consists of a hook (*i.e.*, a function pointer) that guards a call to the libc utility with the arguments passed to the wrapper. The hook must point to a valid security policy (written as a C/C++ function) that determines the policy under which the libc utility can be invoked. For example, the hook for `fopen` can specify the list of files/modes that the application can open. A policy that unconditionally allows `fopen` would simply return `TRUE`. If the hook is `NULL`, *i.e.*, a policy is not provided, then the call to the libc utility is denied.

Figure 6 shows an example of the wrapper for `fopen`. This wrapper uses the file pathname and opening mode to obtain a file descriptor, which is then evaluated against the policy. The use of file descriptors (instead of pathnames) avoids time-of-check to time-of-use vulnerabilities and binds the object being checked with the object that was opened [8]. ESROS2-SCgate wrappers for other primitives must similarly ensure that security policies operate on stable references to the actual

resources being accessed to ensure that the object checked by the policy is the same as the object ultimately used. Security policies are written as C/C++ functions and are arbitrarily expressive. Data from our study in Section 3.2.2 suggests that the policy for a SROS2 application consists of a median of 27 such functions.

Applications load libraries such as `libc` through dynamic linking, which allows ESROS2 to transparently enforce security policies without modifying application code. ESROS2 rewrites the application’s GOT/PLT so that calls to `libc` utilities are redirected to wrappers in ESROS2-SCgate. These wrappers perform policy checks before invoking the underlying `libc` function, transforming the violating configuration shown in Figure 3(d) into the configuration shown alongside, which respects ESROS2-invariant. Consequently, all such operations are subjected to policy enforcement and comply with ESROS2-invariant.

Applications that statically link `libc` bypass ESROS2-SCgate redirection and would therefore violate ESROS2-invariant when issuing system calls, although we did not observe such applications in our study. While our evaluation focused on `libc`, the same approach can be applied to any library that issues raw system calls. In practice, however, we found that libraries such as `libstdc++` rely on `libc` rather than issuing system calls directly.



4 Implementation of ESROS2 using eBPF

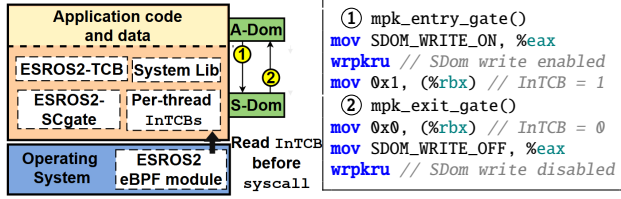
When control passes to the kernel upon a `syscall` instruction, ESROS2 performs an in-kernel check to ensure that the application’s runtime stack is compliant with ESROS2-invariant. Our ESROS2 prototype targets the Linux kernel and uses the extended Berkeley Packet Filters framework (eBPF) [21, 25, 47, 80]. eBPF provides a method to safely and efficiently run sandboxed programs with kernel privileges without requiring changes to kernel source code or having to load untrusted kernel modules. User programs that wish to leverage eBPF must first be compiled down to eBPF bytecode. Several restrictions are placed on the bytecode to ensure that it can be executed safely within the kernel. Once compiled, this bytecode is executed within an in-kernel interpreter (or, increasingly is JITed), allowing it to extend the kernel’s functionality at certain pre-specified hooks within the kernel, such as system call entry points. We now describe how ESROS2 is bootstrapped, and how it enforces ESROS2-invariant.

Bootstrap. ESROS2 checks for the presence of ESROS2-TCB stack frames when it encounters a system call. ESROS2-TCB libraries are dynamically linked against an application, and therefore it is critical to ensure their integrity during application startup. If incorrect/malicious versions of ESROS2-TCB libraries are loaded, then an attacker can easily bypass SROS2, thereby rendering ESROS2 moot. Execution of an

application begins by loading dynamically-linked libraries, including ESROS2-TCB into the program’s address space. Although the process issues several system calls during startup, ESROS2 does not enforce ESROS2-invariant until ESROS2-TCB is loaded. This is because SROS2 has not yet initialized and its security policies are not yet being enforced. Once ESROS2-TCB has loaded, ESROS2’s eBPF module checks the integrity of ESROS2-TCB’s code by computing a hash over the library binaries and checking that they correspond to the correct version. We perform this hash-based checking using an eBPF program that attaches to the `mmap_file` security hook, which is invoked when the dynamic loader calls `mmap` to allocate space for the library in the process address space. The eBPF module notes the address range in which the libraries are loaded. Implementing SHA256 hashing directly in eBPF is challenging because the algorithm is complex and the resulting program cannot be checked by the eBPF verifier. We thus leveraged the `kfunc` wrapper [40, 41] functionality of eBPF. We created a custom kernel function with a `kfunc` wrapper for eBPF to compute the hash, thereby allowing this function to be invoked directly from eBPF just like other eBPF helper functions. Once ESROS2-TCB has loaded, the eBPF module write-protects the global offset/procedure linkage tables and switches to enforcing ESROS2-invariant on all subsequent system calls.

ESROS2-invariant Enforcement. ESROS2 leverages the following observation: To enforce ESROS2-invariant, it suffices to track whether the context of the system call originates in ESROS2-TCB. That is, if the application has invoked ESROS2-TCB and the origin of the system call is from ESROS2-TCB—directly as in Figure 3(a), or indirectly via `libc` as in Figure 3(b)—then the system call may be permitted. This observation allows us to simplify the implementation of ESROS2 by simply performing a bookkeeping of the stack-state of the application using a per-thread InTCB flag. This flag is set to `TRUE` when an application thread enters ESROS2-TCB and is reset to `FALSE` when the ESROS2-TCB returns control back to application logic. Upon a system call, ESROS2’s eBPF module simply consults the calling thread’s InTCB flag to determine whether to allow the system call.

Intra-process Isolation in ESROS2. Given its role, the state of this InTCB flag is security-sensitive. It must therefore be isolated from the application and its value must only be controlled by ESROS2-TCB. We achieve both these objectives by leveraging lightweight intra-process isolation hardware that is now widely available in commodity processors. Our prototype uses Intel MPK [37], although similar hardware is also available on AMD [67], ARM64 [6, 90]. Intel MPK allows lightweight intra-process isolation by allowing process address-spaces to be split into disjoint *domains* (up to 16 domains are allowed). This is accomplished by associating a 4-bit value corresponding to the domain number with each page table entry. Each domain is thus a set of pages, and MPK allows their read/write permissions to be controlled using a



Assume `%rbx` points to `InTCB` of calling thread. `wrpkr` sets `PKRU` to the value of `%eax` (`wrpkr` expects registers `%ecx` and `%edx` initialized to 0 [86], which is not shown in the code above).

Figure 7. The process address space is split into A-Dom and S-Dom. Inter-domain transitions happen via entry/exit gates.

32-bit user-space register called `PKRU`. All the pages belonging to a particular domain can be write-protected by setting `PKRU` to an appropriate value. `PKRU` can be set in user-space using the `wrpkr` or `xrstore` instructions [86].

When `ESROS2` initializes an application, the `ESROS2` eBPF module divides its address space into two domains, A-Dom (for application code) and S-Dom (for system call invocations), as in Figure 7. All application code and data is loaded in A-Dom. `ESROS2-TCB` code and data, including the storage of `InTCB` are loaded into S-Dom. The code/data of `libc` is also dynamically linked into S-Dom. Before transferring control to the main function of the application, `ESROS2`’s eBPF module write-protects the pages in S-Dom.

An application thread that wishes to change the permissions associated with S-Dom must invoke an *entry gate* (`mpk_entry_gate`), which gives write permissions to pages in S-Dom, and sets the `InTCB` flag (stored in S-Dom) of the calling thread to `TRUE` (see Figure 7). A corresponding *exit gate* (`mpk_exit_gate`) sets `InTCB` to `FALSE` and revokes the write permissions of S-Dom pages. We place a call to the entry gate at each entry-point API to `ESROS2-TCB`, and a call to the exit gate at each exit from `ESROS2-TCB`. Note that once an application invokes an API entry-point into `ESROS2-TCB`, there are only two ways in which `ESROS2-TCB` returns control back to the application: (a) when the API returns after completing its task; and (b) if the application has registered a callback with `ESROS2-TCB`, and `ESROS2-TCB` invokes this callback. We handle case (a) by simply inserting a call to `mpk_entry_gate` at the beginning of the `ESROS2-TCB` API call, and `mpk_exit_gate` at the end. To handle case (b), we insert a call to `mpk_exit_gate` just before the callback executes, because the callback transfers control to application logic. When the callback returns, control resumes at the point after the callback (in `ESROS2-TCB`), and we insert a call to `mpk_entry_gate` at these resume points. Fortunately, `SROS2` has a clearly-defined callback interface, and we instrument these interfaces as above (see our code artifact). The calls to entry/exit gates placed in `ESROS2-SCgate` wrappers (shown in Figure 6) serve a similar purpose, as they explicitly authorize the application to invoke a `libc` function after the corresponding policy check has been invoked.

Observe that in our design, application code running in A-Dom can freely invoke any code running in S-Dom. For example, the application can freely invoke `libc` string utilities (e.g., `strlen` and `strcpy`) that are located in S-Dom. `MPK` write-protects S-Dom pages but does not prevent the application from jumping to and executing S-Dom code as long as it does not modify data stored in S-Dom. However, even if the application does jump to and execute S-Dom code, `ESROS2`’s eBPF module will reject system calls originating in S-Dom’s code. This is because the A-Dom to S-Dom transition must happen via the entry gate to set `InTCB`, which eBPF checks before permitting a system call. Thus, system calls from `libc` code are permitted only if the application invokes them from `ESROS2-SCgate` after any policy checks have succeeded.

Asynchronous Primitives. `SROS2` applications rely on asynchronous execution primitives such as callbacks, POSIX signals, C++ exceptions, and `setjmp/longjmp`. We focus on the three most common primitives—signals, callbacks, and exceptions—and show how `ESROS2` preserves execution-domain semantics when handling them. Proper support is necessary because asynchronous events can trigger non-trivial control-flow transfers across domains, potentially bypassing normal domain-switching paths. Since `ESROS2` enforces isolation by updating `PKRU` permissions at domain boundaries, it must explicitly handle such transitions to prevent attackers from circumventing domain protections. Although we do not explicitly support all asynchronous primitives, our design can be extended to them using the same principles.

(1) For *signal handling*, `ESROS2` introduces a universal signal handler. Whenever an application registers a signal handler through signal or sigaction, `ESROS2` intercepts registration at the `libc` layer and replaces the user-specified handler with its own universal handler.

When a signal is delivered, control is first transferred to the universal signal handler rather than directly to the application-defined handler. Before performing the domain transition, the universal handler saves the current execution state, including the program counter and `PKRU` value, in a dedicated per-thread signal stack located within the trusted domain. This avoids relying on the untrusted userspace stack for storing security-critical state. The universal handler then updates the `PKRU` register to revoke write access to the trusted domain, sets the `InTCB` flag to indicate application-domain execution, and invokes the original application-defined signal handler. Once the handler returns, the universal handler restores the saved state from the trusted-domain signal stack, re-establishes the trusted-domain permissions, and restores the `InTCB` flag. Thus signal-induced control-flow transfers and ordinary `ESROS2-SCgate` transitions satisfy the same security invariants. Nested signal delivery occurs when a signal is delivered while a previous signal handler is still executing. The trusted-domain signal stack naturally supports such nested invocations by pushing and popping execution state

on each signal entry and return. In the current implementation, we save the program counter and PKRU value, although additional state can be stored if required.

(2) *ROS2 callbacks* may transition from S-Dom to A-Dom application code. To handle these transitions, we analyzed the SROS2 source code and identified the SROS2 functions responsible for invoking subscription, timer, service, action-client, and action-server callbacks. These callbacks are considered exit points from S-Dom to A-Dom. To handle them, ESROS2 instruments these callback invocation points to specifically include an exit gate that updates the InTCB state before control leaves the trusted component. A full description of the callback dispatch functions identified for each callback type is available in our code artifacts.

(3) *C++ exceptions* can transfer control from trusted code to untrusted code without following the normal wrapper return path. ESROS2 thus intercepts `__cxa_begin_catch`, a C++ runtime function invoked whenever execution enters a catch block. Upon reaching this function, ESROS2 determines whether execution is entering A-Dom or S-Dom code, updates the InTCB state accordingly, and adjusts the PKRU permissions to reflect the target domain. ESROS2 does not require additional state to support nested exceptions. Every caught exception passes through `__cxa_begin_catch`, allowing ESROS2 to reconstruct the correct execution-domain state directly from the current execution context at each catch site without saving and restoring exception-specific state.

5 Security Analysis

This section analyzes the security of ESROS2 against a range of attack vectors, including direct system-call invocation, unauthorized PKRU manipulation, asynchronous control transfers, and previously reported attacks on MPK-based isolation systems [12, 79]. We show that ESROS2 prevents these attacks through a combination of trusted wrappers, Intel MPK, Intel CET, and kernel-level eBPF enforcement.

Invoking System Calls. Attacks can invoke system calls via:

(1) *Direct invocation.* Attempts to invoke system calls directly (e.g., a raw system call) are intercepted and rejected within eBPF, because they would not have modified the application’s security context (i.e., InTCB).

(2) *libc wrappers.* Recall that ESROS2 instruments the GOT and PLT entries of untrusted applications and libraries. Symbols are resolved eagerly at load time, after which GOT entries are write-protected. ESROS2 intercepts all calls to `dlopen`, allowing the same protections to be applied to dynamically loaded libraries. As a result, attackers cannot tamper with GOT or PLT entries to redirect execution to arbitrary code.

The attacker could attempt to redirect control flow to an existing ESROS2-SCgate wrapper. However, ESROS2 leverages Intel CET, which restricts indirect control transfers to valid function entry points. Every ESROS2-SCgate wrapper

entry point has policy validation logic, and attackers cannot therefore jump directly to the system-call instruction or bypass policy validation. Consequently, any system call remains subject to both the ESROS2-SCgate’s policy checks and the kernel-level eBPF enforcement mechanism.

Accessing MPK Gates. ESROS2 adopts ERIM’s [86] approach of performing binary inspection to ensure that no instruction sequence in the untrusted domain can be interpreted as `wrpkru`, `rdpkru` or `xrstor`. In ESROS2, an application’s executable code pages are non-writable, preventing an attacker from injecting new MPK instructions at runtime. Consequently, attackers cannot create arbitrary MPK gates or introduce new mechanisms for modifying PKRU permissions outside of MPK gates. Existing MPK gates reside within ESROS2-SCgate wrappers and are intended to be reached through those wrappers. Therefore, even if an attacker redirects execution to a wrapper containing an MPK gate, the corresponding policy checks will be triggered before a system call is executed. As a result, modifying PKRU permissions does not provide a path around policy enforcement.

Asynchronous Control Transfers. Asynchronous control transfers such as signals, exceptions, callbacks, and non-local jumps introduce additional challenges because they can alter execution flow without following normal call-return semantics. However, as described in Section 4, ESROS2 explicitly handles these mechanisms by introducing dedicated ESROS2-SCgate exit gates at every asynchronous control-transfer point. These gates ensure that the InTCB flag cannot be manipulated to bypass execution-context tracking. Upon an asynchronous transfer, ESROS2-SCgate determines the source and destination domains and performs the appropriate domain transition. Transitions between S-Dom and A-Dom update the InTCB flag and reconfigure PKRU permissions to enforce the correct access rights in the destination domain. For signal handling, ESROS2 maintains a per-thread signal stack in S-Dom and saves and restores the relevant execution state upon signal delivery and return. Attackers therefore cannot exploit asynchronous control transfers to evade domain-transition checks, thereby preserving ESROS2’s guarantees.

Bypassing MPK’s Sandbox. Recent work [12, 79] demonstrates several attacks against MPK-based sandboxes that rely on invoking system calls to manipulate process resources, memory mappings, or trusted components. Such attacks arise from a mismatch between the threat model of in-process isolation systems and the assumptions made by PKU and Linux, allowing attackers to use the kernel as a confused deputy.

ESROS2 is resilient to these attacks because its security guarantees do not rely solely on MPK-based isolation. For a system call to be permitted, it must satisfy both the policy checks enforced by ESROS2-SCgate wrappers and the kernel-level eBPF ESROS2-invariant check. Even if an adversary succeeds in bypassing MPK-based isolation [12, 79], unauthorized system calls are denied by the eBPF enforcement layer unless explicitly authorized through the application’s

ESROS2-SCgate wrapper. Circumventing MPK alone is insufficient to violate ESROS2’s system-call mediation policy. **Control Hijacking.** Once ESROS2-TCB is correctly bootstrapped, application control hijacking attacks cannot bypass policy enforcement. Any control hijack into a S-Dom function that bypasses ESROS2-SCgate will also bypass the MPK call gate, which sets the `InTCB` variable. Without `InTCB=TRUE`, the system call will fail at the eBPF enforcement layer. Control hijacks into A-Dom functions cannot execute system calls without eventually passing through ESROS2-SCgate, where policy checks are applied. Finally, any dynamically loaded libraries (outside of ESROS2-TCB) are handled by intercepting `dlopen`. Such libraries are placed in A-Dom, thereby ensuring that any security-critical operations are routed through ESROS2’s policy enforcement mechanism.

6 Case Studies using ESROS2

We report three case studies with real-world SROS2 application suites to evaluate the following aspects of ESROS2: How effective is ESROS2-invariant as a general security principle? How frequently do applications directly invoke `libc` functionality outside the ESROS2-TCB, and what effort is required to develop ESROS2-SCgate policies? Does ESROS2 prevent behaviors that violate SROS2 security policies? All our experiments were conducted on Ubuntu 22.04.5 with Linux kernel 6.8.0-79 using the Gazebo simulator [28] included with SROS2 Humble Hawksbill, running on a 36-core Intel Xeon Gold 6150 CPU (2.70GHz) with 251GB RAM.

Ⓐ **Robotic Arm.** This study considers an application suite that controls a simulated Franka Emika Panda robotic arm [71], a seven-joint platform widely used in robotics research. The system comprises several cooperating SROS2 binaries that communicate through publish/subscribe mechanisms. Specifically, `static_transform_publisher` [84] anchors the robot in the global reference frame, `robot_state_publisher` [77] publishes link poses of joint states, `ros2_control_node` [69] provides simulated hardware interfaces, and MoveIt’s `move_group` [50] performs motion planning and collision checking. Together, these components enable robot control and exemplify the common SROS2 deployment model in which multiple nodes cooperate to provide system functionality. Like many SROS2 applications, they also rely on `libc` utilities for tasks such as logging and file access. We executed the application suite with ESROS2 in audit mode, where violations of ESROS2-invariant were logged but did not terminate execution. This allowed us to identify violations arising from direct use of `libc` utilities and to derive ESROS2-SCgate policies for utilities that must remain accessible when ESROS2 is deployed in *enforcing mode*, where violations of ESROS2-invariant result in application termination.

Figure 8(a) presents the total number of unique `libc` utilities invoked directly from the application, *i.e.*, those that violate ESROS2-invariant in the absence of ESROS2-SCgate

policies. We have also listed the number of `libc` utilities from various categories with the caveat that some utilities are listed under multiple categories.

We observed that the bulk of the `libc` utilities invoked within the ROS2 binaries correspond to typical file-related operations such as `fopen`, `read`, `fseek`, `fclose`, and `fwrite`, whose primary intent is local file management rather than communication or device control. We confirmed that these operations are benign in intent and are required for the application suite to function properly. We therefore wrote policies to explicitly allow these functions to work with the specific files required for the application to work. The most interesting case that we observed involved `move_group`’s use of the `socket` and `socketpair` utilities from `libc`. These utilities are generally used for network communication, and unrestricted network communication will clearly be in violation of the SROS2 publish/subscribe confinement policies. Upon further analysis, we found `move_group` uses `socket` and `socketpair` during initialization. The initialization parameters are provided via a file, and `move_group` uses Unix-domain sockets with a URL prefixed with `package://` and the path name of the file to read the contents of the local file. After confirming that this operation is also benign and can be permitted, we wrote ESROS2-SCgate policies to permit this and the previously-mentioned file operations with the specific arguments (*e.g.*, file paths) observed during audit-mode execution.

Overall, the security analyst’s effort is proportional to the number of unique `libc` calls invoked by each binary. We had to write one ESROS2-SCgate policy for each of the `libc` utilities for each binary, *i.e.*, 16 for `move_group`, 5 for `robot_state_publisher`, and so on. With these ESROS2-SCgate policies in place, we did not observe any violations of ESROS2-invariant during subsequent operation.

We evaluated the overall runtime overhead introduced by ESROS2 for this case study by measuring the round-trip completion time for a predefined round-trip motion of the robotic arm between a source point and a destination. Across five trials of this experiment, the average latency on a vanilla SROS2 system was 9486.4 ± 1006.6 ms, while the latency was 9880.8 ± 452.8 ms with ESROS2. This corresponds to an average latency overhead of approximately 4.16%.

Ⓑ **Night Vision Image Processing Pipeline.** This pipeline uses three SROS2 applications, with a USB webcam connected to the system as the image source. One of the SROS2 applications communicates with the camera device driver. The SROS2 applications in this suite also use complex packages such as OpenCV and Intel Thread Building Blocks (TBB) for data processing. We saw several more `libc` utilities invoked by these applications compared to the median number of `libc` calls reported in Figure 5.

The first application uses the `v4l2_camera` ROS2 package, which acts as a camera driver using the Video4Linux2 (V4L2) API to stream images from the USB webcam. The

8(a) Robotic Arm		
ROS2 Binary	libc Utilities	
move_group	16	($\mathcal{F}$:12, $\mathcal{D}$:10, $\mathcal{C}$:9)
robot_state_publisher	5	($\mathcal{F}$:5, $\mathcal{D}$:4, $\mathcal{C}$:4)
ros2_control_node	9	($\mathcal{F}$:9, $\mathcal{D}$:7, $\mathcal{C}$:6)
static_transform_publisher	2	($\mathcal{F}$:2, $\mathcal{D}$:1, $\mathcal{C}$:1)
8(b) Night Vision Image Processing		
ROS2 Binary	libc Utilities	
v4l2_camera_node	38	($\mathcal{F}$:14, $\mathcal{D}$:8, $\mathcal{C}$:17)
night_vision	38	($\mathcal{F}$:13, $\mathcal{D}$:7, $\mathcal{C}$:16)
showimage	7	($\mathcal{F}$:3, $\mathcal{D}$:2, $\mathcal{C}$:3)
8(c) Autonomous Navigation using SLAM and Nav2		
ROS2 Binary	libc Utilities	
async_slam_toolbox	11	($\mathcal{F}$:11, $\mathcal{D}$:8, $\mathcal{C}$:8)
behavior_server	4	($\mathcal{F}$:4, $\mathcal{D}$:3, $\mathcal{C}$:3)
bt_navigator	8	($\mathcal{F}$:8, $\mathcal{D}$:7, $\mathcal{C}$:6)
controller_server	4	($\mathcal{F}$:4, $\mathcal{D}$:3, $\mathcal{C}$:2)
lifecycle_manager	2	($\mathcal{F}$:2, $\mathcal{D}$:2, $\mathcal{C}$:2)
planner_server	4	($\mathcal{F}$:4, $\mathcal{D}$:3, $\mathcal{C}$:2)
smoother_server	8	($\mathcal{F}$:8, $\mathcal{D}$:6, $\mathcal{C}$:5)
velocity_smoother	2	($\mathcal{F}$:2, $\mathcal{D}$:2, $\mathcal{C}$:2)
waypoint_follower	4	($\mathcal{F}$:4, $\mathcal{D}$:3, $\mathcal{C}$:2)

Figure 8. libc utilities invoked by different binaries in our case studies. We also provide a breakdown of the sub-categories ($\mathcal{F}$ –File management, $\mathcal{D}$ –Device management, $\mathcal{C}$ –Communication).

second application, `night_vision`, subscribes to the output of `v4l2_camera`, enhances the brightness of images with low light, and then displays the output using `showImage`.

We observed that the `v4l2_camera` package invokes 38 distinct libc functions during execution. This application performs low-level device accesses and therefore uses file-management-related utilities to access system state (CPU, memory, scheduling state) information from Linux `sysfs`. In addition, it uses file I/O operations to append data to log files for auditing purposes. This application uses the `ioctl` libc system call wrapper to communicate with the `Video4Linux2` kernel driver for camera configuration, buffer management, and frame acquisition. It also uses `mmap` to map camera buffers directly into user space to avoid unnecessary data copies. Finally it invokes thread- and scheduling-related libc utilities because camera frame acquisition involves blocking I/O operations, such as waiting for a new frame from the device. To ensure that these blocking operations do not interfere with ROS2 callback execution, the node creates a dedicated POSIX thread (via `std::thread`) that runs independently of the ROS2 executor. This thread continuously captures frames from the camera and publishes them when available, while the ROS2 executor remains free to handle timers, parameter updates, diagnostics, and other callbacks. It also uses `syscall` to invoke the `futex` system call.

The `night_vision` application uses the OpenCV library, which uses Intel TBB for parallel processing. As a result, it executes several image-processing operations (*e.g.*, CLAHE, Gaussian blur, and color mapping) using a pool of up to 16 internal worker threads, which are managed independently of the ROS2 executor. Consequently, the application invokes

multiple thread- and scheduling-related libc functions. In addition, the application uses `mmap` to allocate anonymous memory. It also invokes file-management-related libc APIs for the same reasons discussed previously for the `v4l2_camera` node. The `showimage` application uses 7 libc utilities. Figure 8(b) summarizes the number of libc utilities invoked by the ROS2 applications in this case study.

To measure ESROS2’s overheads for this case study, we measured the frame processing rate of images processed by the application pipeline. Atop a vanilla SROS2 system, the average processing rate was 29.231 frames per second, which fell to 27.249 frames per second atop ESROS2. Thus, ESROS2’s mechanisms resulted in approximately 6.78% reduction of image processing throughput.

© Autonomous Navigation using SLAM and Nav2. This case study involves autonomous navigation of a simulated TurtleBot3 robot [2] by integrating the SLAM toolbox [44] and the ROS2 Navigation2 (Nav2 [45]) framework.

The SLAM binary, `slam_toolbox`, performs simultaneous localization and mapping asynchronously, continuously updating a 2D map even as the robot navigates. The Nav2 stack consists of modular servers (*i.e.*, multiple binaries) that together enable autonomous goal-directed movement. Specifically, `planner_server` generates global paths, while `controller_server` computes velocity commands to follow them. The `velocity_smoother` application performs trajectory refinement while `smoother_server` ensures safe execution. Additional Nav2 servers such as `behavior_server` provide recovery strategies, `waypoint_follower` supports multi-goal missions, and `bt_navigator` orchestrates navigation via a behavior tree. Finally, `lifecycle_manager` ensures that all components are properly initialized and managed.

Figure 8(c) shows that most of the libc utilities invoked by the binaries in this case study only perform file management. Even the large and complex application `async_slam_toolbox` used only a limited set of libc utilities, primarily related to benign file-system activities (*e.g.*, writing out a log). As Figure 8(c) shows, only a modest number of ESROS2-SCgate policies are required even in this case.

For performance evaluation, we measured the round-trip navigation time between two predefined coordinates on a map. We measured the time for five round-trips using both a vanilla SROS2 and an ESROS2 platform, and observed an average increase in round-trip time of 5.25% with ESROS2.

Key Takeaways. By comparing the three case studies, we observe that only a small fraction of reachable libc functions are actually invoked at runtime. For example, static analysis of the `move_group` binary identified 186 reachable libc functions across the binary and its untrusted libraries. However, only 16 of these functions were exercised during execution. This suggests that policy generation is tractable, as even large real-world binaries such as `move_group` and `async_slam_toolbox` invoke only a limited set of libc functions in practice. More

importantly, all communication-related operations are performed through SROS2 APIs rather than direct libc invocations. Consequently, stricter policies can be enforced on application binaries, ensuring that communication occurs exclusively through SROS2 APIs, consistent with the behavior of legitimate SROS2 applications.

7 Benchmark Performance Evaluation

To evaluate the performance impact of ESROS2 on standard benchmarks, we considered the Cedar, Sierra Nevada and Mont Blanc benchmarks from the iRobot suite [38]. This is a standard suite of applications designed specifically to evaluate various aspects of ROS2 performance. Each of these benchmarks consists of several nodes—8 each in Cedar/Sierra Nevada and 9 for Mont Blanc. These nodes interact with each other using publish/subscribe. Data from a publisher node may be consumed by one or more subscriber nodes, with the resulting topology determined by how nodes are matched up for communication based on topic match. In our experimental setup, each application is sandboxed with ESROS2. The hardware platform is the same as the one used in Section 6. We measured two aspects of ESROS2’s performance:

(1) *Application startup time*. During application startup, the steps described in ESROS2’s bootstrap are executed, such as integrity verification of ESROS2-TCB libraries, environment variables, and populating the GOT/PLT with entries corresponding to ESROS2-TCB and ESROS2-SCgate. As a result, delays are added to the startup time of the application. Figure 9 reports the average startup time observed on SROS2 and on ESROS2, measured as the delay between the `execve` call that launches the application and the execution of the first instruction in the application’s main function. Observe that although the multiplicative factor is about $3.2\times\text{--}3.5\times$, the startup time is in milliseconds, with the time going up from 24–25 milliseconds on SROS2 to about 80–89 milliseconds atop ESROS2. The bulk of the time overhead during startup stems from the load-time hash checks used to verify the integrity of ESROS2-TCB libraries.

(2) *End-to-end latency*. We evaluated the runtime impact of ESROS2 by measuring the end-to-end latency from a publishing node to a subscribing node using 100 MB messages. Since some iRobot topologies contain cycles, we made minor modifications to obtain acyclic communication graphs for measurement purposes. We then measured the latency along the longest path in each topology. As shown in Figure 9, ESROS2 introduces only minimal runtime overhead, increasing end-to-end latency by 1.44%, 1.58%, and 5.90% for the Cedar, Sierra Nevada, and Mont Blanc topologies, respectively.

Microbenchmark Evaluation. After an application has completed startup, the primary source of overhead introduced by ESROS2 arises from the execution of entry and exit gates during transitions between A-Dom and S-Dom. To quantify this cost, we used the LMBench [49] suite to microbenchmark

iRobot [38] Benchmark	Startup time (millisec)		Latency (millisec)	
	SROS2	ESROS2	SROS2	ESROS2
Cedar	24.53±0.12	85.93±1.19	739.44±0.02	750.12±0.01
Sierra Nevada	24.87±0.18	80.92±0.12	840.50±0.04	853.74±0.04
Mont Blanc	25.84±0.19	88.28±0.06	1423±0.3	1507±0.9

Figure 9. Performance overhead of ESROS2 on iRobot benchmarks. Results are averages±standard deviations over 6 runs. Startup time measures the interval from `execve` to the first instruction of `main`. Latency measures the time for a 100 MB message to traverse the longest publish/subscribe path in each topology.

Benchmark	SROS2	ESROS2	Overhead (%)
System Call Latencies (μs)			
Null IO	0.591±0.021	0.661±0.047	11.7
stat	1.448±0.049	1.509±0.002	4.2
open close	2.559±0.089	2.686±0.015	5.0
sig install	0.631±0.001	0.637±0.002	10.3
sig handle	1.693±0.079	1.733±0.003	2.4
IPC and Communication Latencies (μs)			
2p/0k ctxsw	4.86±0.13	5.14±0.05	5.8
Pipe	13.98±0.02	14.57±0.09	4.2
AF Unix	22.71±0.29	23.98±0.32	5.6
UDP	35.77±0.05	36.13±0.07	1.0
RPC/UDP	43.43±0.01	43.95±0.05	1.2
TCP	43.19±0.03	43.64±0.02	1.0
RPC/TCP	55.92±0.10	59.44±0.12	6.3
TCP Conn	182.68±0.89	189.58±0.08	3.8
File-System Throughput (ops/s)			
0k Create	81474	80879	0.7
0k Delete	126365	124235	1.7
10k Create	47093	46174	2.0
10k Delete	82653	82064	0.7

Figure 10. Evaluation with LMBench. Overhead denotes the performance overhead of ESROS2 relative to SROS2. For throughput benchmarks, overhead represents throughput degradation.

the performance impact of system-call execution. Figure 10 presents the results. ESROS2 incurs an average overhead of only 3.98% across all 17 evaluated microbenchmarks. Each benchmark was executed 50 times, and the reported results correspond to the average latency or throughput across runs.

8 Related Work

Figure 11 compares ESROS2 against related work. Sandboxing approaches that use `seccomp` filters [15, 30] have used static analysis to derive system-call sets from binaries and automatically generate `seccomp` [22] policies. These techniques have been further extended through phase-based policies [31, 68] and configuration-aware policy generation [27, 32, 66]. All these approaches rely on `seccomp`-BPF to enforce system-call restrictions. Policy decisions are made solely at the system-call interface, and the constrained expressiveness of `seccomp`-BPF policies prevents them from incorporating application execution context. Thus they cannot distinguish whether a system call originates from trusted framework code or potentially compromised application code, unlike ESROS2, which differentiates between S-Dom and A-Dom contexts.

Ptrace- and eBPF-based sandboxing approaches provide expressive, context-sensitive policies and can distinguish system calls originating from S-Dom code from those issued

Property/Sandbox Type	seccomp-BPF	Ptrace	eBPF-based	Model-based	PKU [79]	ESROS2
Context Sensitivity	✗	✓	✓	✓	✓	✓
Supports Async Primitives	N/A	N/A	N/A	✗	✓	✓
Policy Enforcement Location	K	K+U	K	K	U	U
Policy Granularity	Per-syscall	Per-syscall	Per-syscall	Per-syscall/function	Per-syscall	Per-function
Policy Generation	Auto	Semi-Auto	Semi-Auto	Auto	Manual	Semi-Auto
Policy Expressiveness	Constrained	TC	VB	Constrained	TC	TC
Multi-threading support?	✓	◦	✓	◦	◦	✓
Performance Overhead	Minimal	High	High	High	Low	Low
Kernel Attack Surface Reduction	✓	✓	✓	✓	✓	✓
Userspace Attack Surface Reduction	✗	✗	✗	✗	✓	✓
Kernel/Source Modification Needed?	✗	✗	✗	✓	✓	✗

K: Kernel-space enforcement; U: User-space enforcement; ◦: Limited support; TC: Turing-complete; VB: Verifier-Bounded.

Figure 11. Comparison of ESROS2’s design choices against those of other representative sandboxing mechanisms.

by A-Dom code. However, doing so requires recovering the execution context of every system call through stack unwinding to determine compliance with ESROS2-invariant, in a manner somewhat reminiscent of Java’s (now-defunct) stack inspection [88]. When ptrace-based solutions are applied to multithreaded applications, each thread must be monitored by a tracer, while eBPF-based approaches are constrained by verifier restrictions that limit policy complexity. This context-recovery process incurs substantial overhead; we verified that an eBPF prototype that reconstructs call stacks by unwinding the stack using DWARF [20] metadata introduced 100–150% runtime overhead and approximately 325MB of additional memory overhead, and similar costs are expected for ptrace-based approaches. Both approaches rely on the integrity of runtime stack information, which may be manipulated by a compromised application.

There is a large body of work on model-based system-call sandboxing using models from dynamic [26, 42, 89] or static analysis [23, 33, 87]. These approaches construct models of expected system-call behavior and enforce them by recovering the current execution context at runtime using techniques such as stack walking [23, 24] or execution-state tracking [33]. While effective for many applications, applying them to ROS2 is challenging due to its callback-driven, asynchronous, and highly multithreaded execution model, which makes accurate context reconstruction difficult and can lead to complex policies and significant runtime overhead. In contrast, ESROS2 tracks execution domains directly using Intel MPK, eliminating the need for stack inspection or state-machine tracking. This approach naturally supports callback-driven and multithreaded ROS2 applications with lower overhead.

Recent work on PKU sandboxes [79] aims to strengthen MPK-based intra-process isolation with system-call filtering. The key observation is that an application can break out of a PKU domain if it issues certain system calls. PKU sandboxes thus restrict system calls (*e.g.*, using eBPF [79]) that an application can issue from within a PKU domain. While conceptually similar to ESROS2, the objective of such PKU sandboxes is different, and so are the policies, with ESROS2-invariant restricting application behavior in ESROS2. ESROS2-invariant

also has the pleasant side-effect of preventing an application from issuing PKU-sandbox based system calls (Section 5).

More specifically for ROS, the community has identified several security vulnerabilities in early designs, particularly ROS1, which used unauthenticated, plaintext publish/subscribe communication through a centralized broker [16, 48]. ROS2 addresses these limitations by replacing the broker with decentralized DDS-based communication. Numerous efforts have sought to improve communication security in ROS [9, 16, 17, 19, 36, 72, 85, 92, 94], many of which have since been subsumed by SROS2 [18, 91, 93, 94]. Other work has focused on strengthening DDS security itself [43, 92].

Complementary efforts further enhance SROS2 security. Privaros [7] and ROS+AppArmor [4, 5] provide mandatory access control. Like ESROS2, these systems aim to sandbox SROS2 applications. However, policies in these systems are expressed as end-to-end information-flow constraints spanning multiple applications. Privaros and ROS+AppArmor also enforce policies at a coarser granularity, *i.e.*, using mandatory access control within the OS. By contrast, ESROS2’s sandboxing works at a finer level of granularity by restricting the system call behavior of individual SROS2 applications.

9 Summary

Many robotics platforms now use ROS2 because of the ease with which it enables the deployment of distributed robotics applications. SROS2 adds several security features to secure application communication atop ROS2, but its security enforcement is limited to application communication that uses the SROS2 API. Applications can easily bypass SROS2 policies by simply eliding this API and directly interacting with the underlying system. We presented ESROS2, whose key contribution is the identification and enforcement of a simple invariant that prevents applications from bypassing SROS2’s API in interacting with the underlying system. ESROS2 applies to commodity SROS2 binaries, and our evaluation shows that it can enhance SROS2 security while imposing only a modest runtime performance overhead.

Artifacts URL: <https://github.com/Esros2/ESROS2>

References

- [1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, 2005.
- [2] E. Ackerman. Turtlebot inventors tell us everything about the robot. *IEEE Spectrum*, March 2013. <https://github.com/ROBOTIS-GIT/turtlebot3>.
- [3] E. Ackerman. iRobot Launches Create-3, with ROS2 Built In. *IEEE Spectrum*, 2022.
- [4] AppArmor—an effective and easy-to-use Linux application security system. <https://gitlab.com/apparmor/apparmor/wikis/home/>.
- [5] AppArmor and ROS. <http://wiki.ros.org/SROS/Tutorials/AppArmorAndROS>.
- [6] ARM. Permission indirection and permission overlay extensions in Armv8.9-A/Armv9.4-A (version 2.0), 2024. <https://tinyurl.com/ARM-Permission-Overlays>.
- [7] R. Beck, A. Vijeev, and V. Ganapathy. Privaros: A Framework for Privacy-Compliant Delivery Drones. In *ACM Conference on Computer and Communications Security*, 2020.
- [8] M. Bishop. Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux. Technical report, UC-Davis, CSE-95-8, 1995.
- [9] B. Breiling, B. Dieber, and P. Schartner. Secure communication for the robot operating system. In *Annual IEEE International Systems Conference (SysCon)*, 2017.
- [10] C. Canella, M. Werner, D. Gruss, and M. Schwarz. Automating Secomp Filter Generation for Linux Applications. In *ACM Cloud Computing Security Workshop*, 2021.
- [11] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *USENIX/ACM Symposium on Operating System Design and Implementation*, 2006.
- [12] E. Connor, T. McDaniel, J. M. Smith, and M. Schuchard. PKU pitfalls: Attacks on PKU-based memory isolation systems. In *USENIX Security Symposium*, 2020.
- [13] National Vulnerability Database. CVE-2024-38923. <https://nvd.nist.gov/vuln/detail/CVE-2024-38923>, 2024.
- [14] Data Distribution Service. <https://www.omg.org/spec/DDS/1.4>.
- [15] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis. sysfilter: Automated System Call Filtering for Commodity Software. In *International Symposium on Research in Attacks, Intrusions and Defenses*, 2020.
- [16] B. Dieber, B. Breiling, S. Taurer, S. Kacianka, S. Rass, and P. Schartner. Security for the Robot Operating System. *Robotics and Autonomous Systems*, 98, 2017.
- [17] B. Dieber, S. Kacianka, S. Rass, and P. Schartner. Application-level security for ROS-based applications. In *Intelligent Robots and Systems, 2016 IEEE/RSJ International Conference on*, 2016.
- [18] B. Dieber, R. White, S. Taurer, B. Breiling, G. Caiazza, A. Cortesi, and H. Christensen. Penetration testing ROS. *Robot Operating System (ROS): The Complete Reference (Volume 4)*, 2019.
- [19] R. Doczi, F. Kis, B. Suto, V. Poser, G. Kronreif, E. Josvai, and M. Kozlovsky. Increasing ROS 1.x communication security for medical surgery robot. In *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2016.
- [20] DWARF Debugging Information Format Version 5, 2017. <https://dwarfstd.org/doc/DWARF5.pdf>.
- [21] eBPF documentation. <https://docs.ebpf.io/>.
- [22] J. Edge. A seccomp overview, 2015. <https://lwn.net/Articles/656307/>.
- [23] H. H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, 2005.
- [24] H. H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, 2003.
- [25] M. Fleming. Linux Socket Filtering aka Berkeley Packet Filter (BPF). <https://www.kernel.org/doc/Documentation/networking/filter.txt>.
- [26] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *IEEE Symposium on Security and Privacy*, 1996.
- [27] A. J. Gaidis, V. Atlidakis, and V. P. Kemerlis. Sysxchg: Refining privilege with adaptive system call filters. In *ACM Conference on Computer and Communications Security*, 2023.
- [28] Setting up a robot simulation (Gazebo). <https://docs.ros.org/en/humble/Tutorials/Advanced/Simulators/Gazebo/Gazebo.html>.
- [29] B. Gerky. ROS, the Robot Operating System, is growing faster than ever, celebrates 8 years. *IEEE Spectrum*, Dec 2015. <https://spectrum.ieee.org/automaton/robotics/robotics-software/ros-robot-operating-system-celebrates-8-years>.
- [30] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *International Symposium on Research in Attacks, Intrusions and Defenses*, October 2020.
- [31] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis. Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium*, August 2020.
- [32] S. Ghavamnia, T. Palit, and M. Polychronakis. C2C: Fine-grained Configuration-driven System Call Filtering. In *ACM Conference on Computer and Communications Security*, 2022.
- [33] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *11th Annual Network and Distributed Systems Security Symposium*, 2004.
- [34] I. Goldberg, D. A. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *USENIX Security Symposium*, 1996.
- [35] Object Management Group. About the Data Distribution Service Specification V1.4. <https://www.omg.org/spec/DDS/About-DDS/>.
- [36] J. Huang, C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan, and G. Rosu. RosRV: Runtime verification for robots. In *International Conference on Runtime Verification*, 2014.
- [37] Intel. Intel-64 and IA-32 architectures software developer's manual, 2018.
- [38] iRobot ROS2 performance evaluation framework. <https://github.com/irobot-ros/ros2-performance>.
- [39] S. Kate, Y. Gao, S. Feng, and X. Zhang. ROSCallBaX: Statically Detecting Inconsistencies In Callback Function Setup of Robotic Systems. In *ACM International Conference on the Foundations of Software Engineering*, 2025.
- [40] BPF Kernel Functions (kfuncs). <https://docs.kernel.org/bpf/kfuncs.html>.
- [41] eBPF Documentation—KFuncs (Linux). <https://docs.ebpf.io/linux/kfuncs/>.
- [42] W. Lee, S. Stolfo, and K. Mok. A data mining framework for building intrusion detection models. In *IEEE Symposium on Security and Privacy*, 1999.
- [43] Y. Liu, Y. Guan, X. Li, R. Wang, and J. Zhang. Formal analysis and verification of DDS in ROS2. In *16th ACM/IEEE International Conference on Formal Methods and Models for System Design*, 2018.
- [44] S. Macenski and I. Jambrecic. Slam toolbox: Slam for the dynamic world. *Journal of Open Source Software*, 6(61), 2021. https://github.com/SteveMacenski/slam_toolbox.
- [45] S. Macenski, F. Martín, R. White, and J. Ginés Clavero. The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.
- [46] V. Mayoral-Vilches, R. White, G. Caiazza, and M. Arguedas. SROS2: Usable Cyber Security Tools for ROS 2. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2022.

- [47] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for packet capture. In *USENIX Winter Conference*, 1993.
- [48] J. McClean, C. Stull, C. Farrar, and D. Mascareñas. A Preliminary Cyber-Physical Security Assessment of the Robot Operating System (ROS). In *Unmanned Systems Technology XV*, volume 8741. International Society for Optics and Photonics, 2013.
- [49] L. McVoy and C. Staelin. Lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, 1996.
- [50] MoveIt Motion Planning Framework for ROS 2. <https://github.com/moveit/moveit2>.
- [51] National Vulnerability Database. CVE-2019-5063. <https://nvd.nist.gov/vuln/detail/CVE-2019-5063>, 2019.
- [52] National Vulnerability Database. CVE-2019-5064. <https://nvd.nist.gov/vuln/detail/CVE-2019-5064>, 2019.
- [53] National Vulnerability Database. CVE-2024-25198. <https://nvd.nist.gov/vuln/detail/CVE-2024-25198>, 2024.
- [54] National Vulnerability Database. CVE-2024-25199. <https://nvd.nist.gov/vuln/detail/CVE-2024-25199>, 2024.
- [55] National Vulnerability Database. CVE-2024-30961. <https://nvd.nist.gov/vuln/detail/CVE-2024-30961>, 2024.
- [56] National Vulnerability Database. CVE-2024-37861. <https://nvd.nist.gov/vuln/detail/CVE-2024-37861>, 2024.
- [57] National Vulnerability Database. CVE-2024-37862. <https://nvd.nist.gov/vuln/detail/CVE-2024-37862>, 2024.
- [58] National Vulnerability Database. CVE-2024-37863. <https://nvd.nist.gov/vuln/detail/CVE-2024-37863>, 2024.
- [59] National Vulnerability Database. CVE-2024-38922. <https://nvd.nist.gov/vuln/detail/CVE-2024-38922>, 2024.
- [60] National Vulnerability Database. CVE-2024-41644. <https://nvd.nist.gov/vuln/detail/CVE-2024-41644>, 2024.
- [61] National Vulnerability Database. CVE-2024-41645. <https://nvd.nist.gov/vuln/detail/CVE-2024-41645>, 2024.
- [62] National Vulnerability Database. CVE-2024-41646. <https://nvd.nist.gov/vuln/detail/CVE-2024-41646>, 2024.
- [63] National Vulnerability Database. CVE-2024-41648. <https://nvd.nist.gov/vuln/detail/CVE-2024-41648>, 2024.
- [64] National Vulnerability Database. CVE-2025-4638. <https://nvd.nist.gov/vuln/detail/CVE-2025-4638>, 2025.
- [65] National Vulnerability Database. CVE-2025-53644. <https://nvd.nist.gov/vuln/detail/CVE-2025-53644>, 2025.
- [66] S. Pailoor, X. Wang, H. Shacham, and I. Dillig. Automated policy synthesis for system call sandboxing. *Proceedings of the the ACM on Programming Languages (OOPSLA)*, 4, November 2020.
- [67] S. Park, S. Lee, and T. Kim. Memory Protection Keys: Facts, Key Extension Perspectives, and Discussions. *IEEE Security and Privacy*, 21(3), October 2023.
- [68] V. Rajagopalan, K. Kleftogiorgos, E. Göktas, J. Xu, and G. Portokalidis. SysPart: Automated temporal system call filtering for binaries. In *ACM Conference on Computer and Communications Security*, 2023.
- [69] ROS2 control. https://github.com/ros-controls/ros2_control.
- [70] Transparency Market Research. Robot Operating System Market - Snapshot, 2018. <https://www.transparencymarketresearch.com/robot-operating-system-market.html>.
- [71] RoboDK. Robot Simulation Software—Franka Emika Panda robot. <https://robodk.com/robot/Franka/Emika-Panda>.
- [72] F. J. Rodríguez-Lera, V. Matellán-Olivera, J. Balsa-Comerón, Á. M. Guerrero-Higuera, and C. Fernández-Llamas. Message Encryption in Robot Operating System: Collateral Effects of Hardening Mobile Robots. *Frontiers in ICT*, 5, 2018.
- [73] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security*, 15, 2012.
- [74] ROS.org—Powering the World’s Robots. <https://www.ros.org>.
- [75] ROS 2—ROS 2 documentation, the latest version of the robot operating system. <https://index.ros.org/doc/ros2/>.
- [76] ROS2.org: ROS packages for Humble. https://repo.ros2.org/status_page/ros_humble_default.html.
- [77] ROS.org robot state publisher. https://github.com/ros/robot_state_publisher.
- [78] The Real-time Publish-Subscribe Protocol DDS Interoperability Wire Protocol. <https://www.omg.org/spec/DDS-RTSP/2.3/PDF>.
- [79] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *31st USENIX Security Symposium*, August 2022.
- [80] J. Schulist, D. Borkmann, and A. Starovoitov. A thorough introduction to eBPF, 2017. <https://lwn.net/Articles/740157/>.
- [81] K. Scott. 2025 ROS Metrics Report. <https://discourse.openrobotics.org/t/2025-ros-metrics-report/52575>, 2025.
- [82] A. Silberschatz, G. Gagne, and P. B. Galvin. *Operating System Concepts*. Addison-Wesley, 10th edition, 2025.
- [83] Setting up security—ROS2 documentation. <https://docs.ros.org/en/rolling/Tutorials/Advanced/Security/Introducing-ros2-security.html>.
- [84] ROS.org: tf2_ros static_transform_publisher. https://wiki.ros.org/tf2_ros.
- [85] R. Toris, C. Shue, and S. Chernova. Message authentication codes for secure remote non-native client connections to ROS enabled robots. In *IEEE International Conference on Technologies for Practical Robot Applications*, 2014.
- [86] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security Symposium*, 2019.
- [87] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.
- [88] D. Wallach and E. Felten. Understanding Java Stack Inspection. In *IEEE Symposium on Security and Privacy*, 1998.
- [89] C. Warrander, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, 1999.
- [90] M. Weidmann. ARM Developer Community—Future Architecture Technologies: POE2 and vMTE, October 2025. ARM Developer Blog.
- [91] R. White, G. Caiazza, H. Christensen, and A. Cortesi. SROS1: Using and developing secure ROS1 systems. In *Robot Operating System (ROS)*. Springer, 2019.
- [92] R. White, G. Caiazza, C. Jiang, X. Ou, Z. Yang, A. Cortesi, and H. Christensen. Network Reconnaissance and Vulnerability Excavation of Secure DDS Systems. In *IEEE European Symposium on Security and Privacy Workshops*, 2019.
- [93] R. White, D. Christensen, I. Henrik, and D. Quigley. SROS: Securing ROS over the Wire, in the Graph, and through the Kernel. In *IEEE-RAS International Conference on Humanoid Robots*, 2016.
- [94] R. White, H. Christensen, G. Caiazza, and A. Cortesi. Procedurally provisioned access control for robotic systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2018.

Details on Callbacks in SROS2

In ROS2, a callback is a function whose scheduling and execution are managed by ROS2’s main *executor*. ROS2 callbacks fall under several categories: (1) subscription callbacks, which are used to receive and process topic-labeled data; (2) timer callbacks; (3) service and client callbacks, which are used to handle service requests on the server-side and client-side in the client/server setup of ROS2. The executor manages,

```

static const char *fopen64_allowed_paths[] = {
    // ament index: package index
    "/opt/ros/humble/share/ament_index/resource_index/packages/urdf_parser_plugin",
    "/opt/ros/humble/share/ament_index/resource_index/packages/sdformat_urdf",
    "/opt/ros/humble/share/ament_index/resource_index/packages/urdf",

    // ament index: plugin discovery
    "/opt/ros/humble/share/ament_index/resource_index/urdf_parser_plugin__pluginlib__plugin/sdformat_urdf",
    "/opt/ros/humble/share/ament_index/resource_index/urdf_parser_plugin__pluginlib__plugin/urdf",
}; // Each of these file accesses must be permitted. We call each a resource access check, and there are 5 in this example.

bool fopen64_policy(const char *path, const char *mode) {
    if (strcmp(mode, "r") != 0) // Allow exclusively read-only opens
        return false;

    for (size_t i = 0;
         i < sizeof(fopen64_allowed_paths) / sizeof(fopen64_allowed_paths[0]);
         i++) {
        if (path_matches(path, fopen64_allowed_paths[i])) {
            return true;
        }
    }
    return false;
}

```

Figure 12. Policy for fopen64 of the robot_state_publisher application from the Robotic Arm case study in Section 6.

schedules, and invokes these callbacks on behalf of the application. ROS2 offers both single-threaded and multi-threaded variants of the executor.

ESROS2 uses the rule-of-thumb that callbacks contain application-provided code and must therefore not be permitted to directly execute system calls. Callbacks also represent the situation wherein code in ESROS2-TCB (running in S-Dom) transfers control to application logic (running in A-Dom). In the ROS2 code base (rclcpp in particular), the functions that invoke the callbacks are almost entirely implemented as C++ template functions, defined in headers and instantiated at compile time. ESROS2 must ensure that `mpk_exit_gate` is executed before these functions are called (so that S-Dom is write-protected before the callback is invoked), and that `mpk_entry_gate` is executed once the execution of the callback logic is completed, and control returns to ESROS2-TCB. We now detail the placement of these gates for the categories of callbacks discussed above:

(1) Subscription callbacks. In ROS2, the DDS middleware layer is responsible for message delivery after matching topic names. When ROS2's executor receives a new message from DDS, it invokes a function called `execute_subscription`, which in turn calls C++ templates that are instantiated as part of application logic (via a function called `handle_message`). In ESROS2-TCB, we place a `mpk_exit_gate` call just before the call to `handle_message`, thus write-protecting S-Dom. Control returns to ESROS2-TCB just following this call to `handle_message` after the callback completes, and we place a call to `mpk_entry_gate` to transition from A-Dom to S-Dom.

(2) Timer callbacks. Applications can register code that will be executed when a timer expires. Upon timer expiry, ROS2's executor invokes a function called `execute_any_executable`, which in turn invokes the application-provided callback via a

function called `execute_callback`. As with subscription callbacks, we guard `execute_callback` with the `mpk_exit_gate` gate before and `mpk_entry_gate` after the call.

(3) Service and client callbacks. ROS2 can also be configured to work in client/server mode. When an application is configured as a server, service callbacks are application-registered functions that handle incoming requests from clients. Similarly, if an application is configured as a client, a client callback is application code that allows it to process replies received from a server. With service callbacks, the rclcpp function called `execute_service` invokes the application-registered callback via `handle_request`, the call to which is guarded with an exit gate/entry gate call pair. In the case of client callbacks, the corresponding rclcpp function called `execute_client` invokes the application callback via a call to `handle_response`, which is guarded by the `mpk_exit_gate` and `mpk_entry_gate` gate pair.

In summary, ROS2's clean interfaces to execute callbacks allows us to precisely identify transition points from ESROS2-TCB to application logic, and guard these transitions between S-Dom and A-Dom with suitable gates.

Example of ESROS2-SCgate Policy

This section provides more details on the ESROS2-SCgate policies for the Franka Emika Panda Robotic Arm case study. Our goal is to give an idea of the effort involved in writing policies for ESROS2-SCgate.

Figure 8 of the paper shows the number of libc calls invoked directly from various ROS2 applications in this case study as we ran the applications. Each of these calls succeeds only if the policy registered for the corresponding libc call succeeds. Figure 12 illustrates the function implementing the security policy check for the calls to `fopen64`, one of the file

libc Utility	# Checks	libc Utility	# Checks
move_group		robot_state_publisher	
fclose	64	fclose	4
fcntl	4	fopen	1
fgetc	37	fopen64 ←	5
fopen	39	fwrite	1
fopen64	28	read	5
fread	38	ros2_control_node	
fseeko64	37	fclose	147
ftello64	37	fgetc	92
fwrite	2	fopen	94
mmap	4	fopen64	58
open	32	fread	105
read	29	fseeko64	108
sigaction	1	ftello64	108
socket	1	fwrite	1
socketpair	1	syscall	1
syscall	1	static_transform_publisher	
		fwrite	1
		syscall	1

Figure 13. Number of resource access checks required in the ESROS2-SCgate policies for each libc utility invoked by the ROS2 applications in the Robotic Arm case study.

utilities invoked by the `robot_state_publisher` application. The policy sanity checks the input arguments to `fopen64`, in this case to check that the file(s) are opened in read-only mode, and that only the set of five files permitted by the security analyst can be opened. Calls to utilities like `sizeof` and `strcmp` are directly resolved to the libc code in S-Dom.

In this example, we say that `fopen64` requires five resource access checks, one each corresponding to the file (and the access mode) opened. Figure 13 presents the number of such resource access checks required for each of the libc utilities accessed by all of the applications in this case study. For example, Figure 13 shows that `move_group` accesses a total of 16 libc utilities, and `fopen64` requires 28 access checks.

The case illustrated in the policy listing in Figure 12 is marked. Observe that the applications also accessed the generic `syscall` utility to make system calls. We found that the applications used this utility to obtain thread information by invoking `SYS_gettid`. It is unclear why these applications chose this approach instead of `gettid`. Nevertheless, we permitted it to allow the applications to function, and this is indicated in the single resource access check against the `syscall` utility.

The required resource access checks can be provided *a priori* by the security analyst, who declares what is permissible. Alternatively, a site security analyst unfamiliar with the resources required by an application can run ESROS2-SCgate in auditing mode in a sandboxed environment, observing and vetting the resources accessed, and then including a rule for each resource accessed. A *prima facie* look at the number of resource access checks for `move_group` and `ros2_control_node` in Figure 13 would lead the reader to conclude that several tens to hundreds of resource access checks are required for some libc utilities. However, our experience (with this and other case studies) suggests that most of the resource access checks correspond directly to resource requirements specified in the configuration files of ROS2 applications. For example, all the file path names accessed by the file-related libc utilities in the ROS2 applications in this case study were declared in the configurations of the applications, and the analyst can directly use the file path names (after vetting them) in the resource access checks.

The full set of resource access checks for the ESROS2-SCgate policies for each libc utility invoked by the ROS2 applications (for all three of our case studies) is available in the artifact repository associated with the paper at <https://github.com/Esros2/ESROS2>