# Taking 5G RAN Analytics and Control to a New Level

Xenofon Foukas, Bozidar Radunovic, Matthew Balkwill, Zhihua Lai
Microsoft
Cambridge, United Kingdom

## ABSTRACT

Open RAN, a modular and disaggregated design paradigm for 5G radio access networks (RAN), promises full programmability through the O-RAN RAN Intelligent Controller (RIC). However, due to latency and safety challenges, the telemetry and control provided by the RIC is mainly limited to higher layers and higher time scales ($> 10ms$), while also relying on predefined service models which are hard to change. We address these issues by proposing Janus, a fully programmable monitoring and control system, specifically designed with the RAN idiosyncrasies in mind, focused on flexibility, efficiency and safety. Janus builds on eBPF to allow third-parties to load arbitrary codelets directly in the RAN functions in a provably safe manner. We extend eBPF with a novel bytecode patching algorithm that enforces codelet runtime thresholds, and a safe way to collect user-defined telemetry. We demonstrate Janus' flexibility and efficiency by building 3 different classes of applications (17 applications in total, most not achievable on the existing RIC) and deploying them on a 100MHz 4x4 MIMO 5G cell without affecting the RAN performance.

## 1 INTRODUCTION

A key transformation of the Radio Access Network (RAN) in 5G is the migration to an Open RAN architecture, that sees the RAN functions virtualized (vRAN) and disaggregated. This approach fosters innovation by allowing vendors to come up with unique solutions for different components at a faster pace. Furthermore, a new Open RAN component, called the Radio Intelligent Controller (RIC) [27, 31], allows 3rd parties to optimize the network by building data-driven, vendor-agnostic monitoring and control applications [9, 20] over open interfaces standardized by O-RAN [19].

Despite this compelling vision, the opportunity for innovation largely remains untapped for two main reasons. First, the RAN network functions can generate huge volumes of data at a high frequency. Capturing, transferring and processing the data for developing novel RIC applications can put a strain on compute and network capacity. To overcome this, a conventional approach, standardized by 3GPP [13, 14], defines a small set of aggregate Key Perfomance Indicators (KPIs) collected every few seconds or minutes. The O-RAN RIC extends this idea by providing a new set of standardized aggregate KPIs and data sources [25]. Each KPI is defined through a service model (a form of a static API that is part of a RIC agent that is embedded in the vRAN functions [24]) and prescribes what data can be collected and at which granularity. However, this approach is slow to evolve and doesn't scale well. Anyone who has a use case that doesn't fit into the existing service models, needs to specify a new service model with a different set of KPIs. They then need to work with a selected RIC and RAN vendor to add support for this service model and go through a lengthy standardization process, where all O-RAN vendors must be convinced to support it and implement it.

Second, many key RAN operations, like user radio resource scheduling and power control must be completed within a deadline, typically ranging from a few tens of $\mu$s to a few ms. To meet the deadlines, any related control logic and inference must run inline inside the vRAN functions, rather than on the RIC, which has been designed to deal with time-scales > 10ms [78]. The existing RIC approach deals with this issue by specifying service models tailored to specific use cases, each with a supported set of policies (e.g., choose one out of $N$ available radio resource scheduling algorithms). However, as in the case of data collection, this approach does not scale, since it does not allow the flexible introduction of new control and inference algorithms. Furthermore, the real-time nature of many vRAN operations means that any new functionality added in order to support a new service model must be completed within the processing deadline of the vRAN function, since a deadline violation may cause performance degradation [45] or even crash a vRAN (as we show in §7.2). This makes RAN vendors reluctant to add new features and service models.

To address these problems we propose Janus, a system that provides dynamic monitoring and control vRAN functionality. Janus extends the RIC by allowing operators and trusted third-parties to write their own telemetry, control and inference pieces of code (we call them *codelets*) that can be deployed at runtime at different vRAN components, without any assistance from vendors and without disrupting the vRAN operation. The codelets are typically executed inline and on vRAN critical paths, allowing them to get direct access to all important internal raw vRAN data structures, to collect arbitrary statistics and to make real-time inference and control decisions.

While Janus significantly enhances the RIC capabilities, by decoupling the vRAN implementation from the data collection and control operations, it also comes with its own set of challenges. The first has to do with flexibility. It is unclear how much and which RAN monitoring data is required to build useful apps and what control knobs should be exposed to codelet developers. We solve this by identifying key locations

and interfaces (we call them *hooks*) within the standard vRAN architecture that provide rich and diverse data from a few collection points, as well as allow us to unlock a wide range of real-time control applications. We also build a toolchain that allows developers to define arbitrary output data structures to ship the collected data to the RIC. Finally, we identify a small set of carefully curated control functions (we call them *actuator functions*) that can be called by codelets loaded in control hooks, to alter the RAN behavior in real-time (e.g. change power, resource block allocation, etc). We show in §4 that O-RAN service models can be implemented as Janus codelets, and can also enable fast and efficient control and inference operations (e.g., radio resource allocation and interference detection), not possible using the O-RAN RIC.

The second challenge is about safety of execution. While codelets are provided by trusted parties, they can still have errors and inefficiencies in terms of invalid memory accesses or high execution times, leading to corruption of data, violation of real-time deadlines and ultimately, to the crash of the vRAN functions. We solve this challenge, by providing a sandboxed execution environment based on eBPF [3, 92], which solves a similar problem in the Linux kernel [2]. Codelets are written in C and are compiled into eBPF bytecode. The eBPF bytecode runs inside an eBPF virtual environment inlined in the vRAN's control and data path, and has direct access to selected internal RAN data structures and control functions. Prior to loading a codelet, the eBPF execution environment statically verifies the bytecode [48, 92] and only allows codelets that are safe in terms of memory accesses to run.

We further extend this model to tailor it to the vRAN requirements. We introduce hard, $\mu$s-level control in the execution latency of codelets through a novel eBPF bytecode patching mechanism that preempts a codelet that exceeds a certain runtime threshold. Furthermore, we extend the static verification process of codelets to cover the newly introduced flexible output data structures and we provide several optimizations to ensure a non-preemptible design in the fast path (where Janus codelets run), minimizing Janus' impact to the performance of the vRAN. Finally, we integrate Janus with a commercial 5G vRAN stack from CapGemini [35] (based on the Intel FlexRAN reference design [57]) and with the open source 4G/5G stack of OpenAirInterface (OAI) [26].

In summary, we make the following contributions:

- We propose the first safe and programmable framework for dynamically introducing flexible monitoring and control capabilities to vRAN functions with minimal computational overhead (§3). We illustrate its functionality by developing new telemetry, control and inference applications on top of it (17 applications in total) (§4).
- We propose and build mechanisms for enforcing codelet execution runtimes and for safe data collection, to ensure that the vRAN meets its safety and latency requirements (§5).

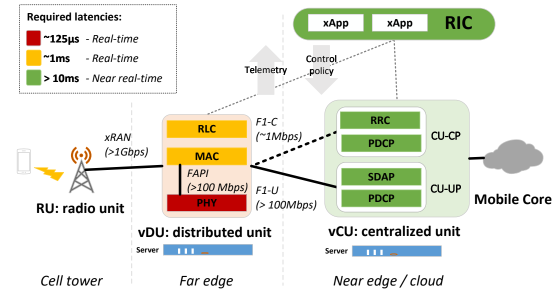



**Figure 1: High-level vRAN architecture and processing and throughput requirements of vRAN functions.**

- We present a concrete and optimized implementation of Janus (§6) and perform a thorough evaluation (§7).

We hope that Janus will be integrated with the O-RAN RIC controller in the future, to further enhance the Open RAN architecture towards the ultimate goal of observability, programmability and automation.

# 2 BACKGROUND & MOTIVATION

## 2.1 vRAN Architecture

The 5G RAN consists of a number of *layers*, illustrated in Fig 1 (e.g., PHY, MAC, RLC). Each layer is responsible for a distinct set of control and/or data plane operations. For example, the PHY is responsible for the signal processing and the MAC for the real-time scheduling of radio resources among the User Equipments (UEs). The layers are distributed among three network functions called the Radio Unit (RU), the Distributed Unit (DU) and the Centralized Unit (CU), which is further broken down into a control plane and user-plane component (CU-CP and CU-UP). The RU is typically ASIC or FPGA-based, while, the CU and the DU are virtualized (i.e., vCU and vDU) and are running on commodity hardware with general purpose processors and accelerators [56, 90]. Different components and layers have different latency requirements(c.f. [32]) and generate events and data at different rates, as shown in Fig 1.

The communication between the vRAN components is achieved through open interfaces specified by standardization bodies like O-RAN [19] and the Small Cell Forum [41], and programmability is facilitated through a near real-time RIC [47]. Network operators install applications (xApps in the O-RAN terminology) on the RIC to collect data and leverage it for inference, closed loop network optimization or to report issues in near real-time (> 10ms). The data collection and control of the vRAN components is facilitated through *service models* that are embedded in the vRAN functions by vendors and define the type and frequency of data reporting for each xApp and a list of control policies that the RIC can enforce.

## 2.2 vRAN programmability limitations

The initial focus of RIC use cases has been on self-optimizing networks, anomaly detection and coarse grained radio resource allocation [60, 75, 78, 85]. In such use cases, significant network events and control decisions occur at a low rate (10s to 100s per second). This allows xApps to collect all the required telemetry, perform inference and tune the vRAN functions through a pre-determined set of control policies. Unfortunately, this approach has some important limitations:
**Data volume limitations:** Many applications like localization [62], channel estimation [66, 71], interference detection [63] and beamforming [72] require uplink IQ samples from the PHY. Transporting all IQ samples to the RIC is infeasible [1]. The current RIC design overcomes this problem by specifying the data required in terms of type and frequency in the service model of each xApp (e.g., as in [37, 38]). The form of data and any required pre-processing (e.g., sub-sampling vs. averages) depends on the service model, posing a serious limitation to interoperability, since vRAN vendors must implement and support each proprietary service model.
**Real-time limitations:** Some vRAN control loops (e.g. UE radio resource allocation and power control) have tight time constraints (10s of $\mu$s to a few ms). Unfortunately, such time constraints cannot be met by the current RIC design, that has an expected latency $>$ 10ms [78]. As in the case of telemetry, xApps overcome this issue by using a set of policies offered by service models, which can run inline inside the vRAN functions. However, this approach doesn't scale as the number of available policies increases. For example, several control algorithms have been proposed for network slicing (e.g., [42, 49, 52, 61, 74, 87]), each tailored to a specific use case. Implementing such algorithms as part of a service model becomes extremely difficult, since all RAN vendors must adopt them.

## 2.3 vRAN programmability requirements

Based on the aforementioned limitations, we argue that a new type of solution is needed for unlocking the true RIC capabilities. Such a solution should meet the following requirements:
(1) Flexible telemetry, where trusted developers can access raw vRAN data and choose the type, frequency and granularity of the exported data, based on the requirements of their application and the limitations of the infrastructure.
(2) Capability to implement arbitrary control and inference logic that can run inline inside the RAN functions in real-time.
(3) A safe execution environment, that guarantees that any (trusted) code that is running inside the vRAN functions will not crash the vRAN by performing invalid memory accesses or by leading to real-time processing deadlines violations.

[1] It requires more than 5 Gbps per cell for 100 MHz $4 \times 4$ MIMO

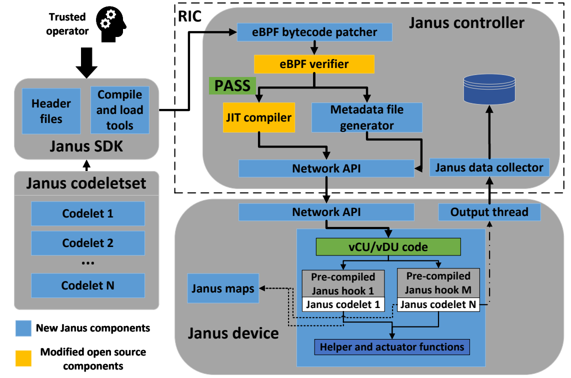



**Figure 2: The high-level architecture of Janus.**

# 3 JANUS OVERVIEW

To overcome the aforementioned limitations, Janus introduces an inline code execution framework that allows the dynamic loading of custom telemetry or control/inference code in a sandboxed environment in the vRAN functions.

## 3.1 Inline code execution framework

The high-level architecture of Janus is illustrated in Fig 2. We next describe the main components.
**Janus device:** A Janus device is any vRAN component (i.e., a vCU or vDU process) that allows execution of custom code. Janus executes the custom code inside an eBPF VM instantiated in userspace [59]. We introduce Janus call points, or *hooks*, at selected places in vRAN functions, at which custom eBPF code can be invoked. The invocation is inlined with the vRAN code and gives the eBPF code read-only access to a selected internal vRAN context and helper functions, which includes various 3GPP-defined data structures (see Table 1 and §3.2). A custom code can be loaded and unloaded dynamically from a Janus device, at runtime, without affecting the device's performance.We opted for eBPF as the sandboxing technology because it is inlined, fast, supports writing codelets in a high level language (C) and provides static code verification. Other approaches, such as Sandbox2 [50] and SAPI [51], run custom code in separate processes and the IPC latency is too high. WebAssembly [53] is inlined, but its lack of static verification can lead to memory violation issues [64].
**Janus codelets:** A Janus *codelet* is a custom code that can be deployed at a single hook on a Janus device at runtime. Developers write codelets in C and compile them into eBPF bytecode using the eBPF compiler. Similar to any eBPF program, a Janus codelet must be statically verifiable (e.g., code must introduce memory checks and can only have bounded loops). Any operations required by the codelet that could be potentially unsafe (e.g., accessing memory outside of the region the codelet is allowed to access) can only be performed

| Hook point | vRAN function(s) | Context description |
|---|---|---|
| Raw UL IQ samples | vDU | Capture uplink IQ samples sent by RU to vDU through xRAN 7.2 interface [22] |
| FAPI interface | vDU | Capture scheduling and data plane packets exchanged between MAC and PHY layers [41] |
| RLC | vDU | Capture information about buffers of mobile devices and RLC mode/parameters [15] |
| F1/E1/Ng/Xn interfaces | vCU/vDU | Capture control/data-plane messages exchanged between 3GPP interfaces of vCU/vDU/5G core [10–12, 17] |
| RRC | vCU | Capture RRC messages exchanged between mobile devices and the base station [16] |

**Table 1: Janus monitoring hooks introduced in commercial-grade vCU/vDU network functions and OpenAirInterface.**

through a set of white-listed *helper* functions. A codelet does not keep any state between two invocations. All state must be stored in an external location called a *map*. A codelet sends its telemetry data through a special map to a Janus output thread running at the device, which forwards it to the Janus controller. A *codeletset* is an ensemble of codelets that operate across multiple Janus hooks of a Janus device and coordinate with very low latency through shared maps. Codelets across devices can coordinate through a controller if needed.

**Janus controller and SDK:** The Janus controller is responsible for controlling the Janus devices and codeletsets. In the context of O-RAN, it can be implemented as a RIC xApp. Developers upload their codeletsets to the controller, with load/unload instructions for one or more Janus devices. Before the controller allows a codeletset to be loaded, it verifies safety and termination of each codelet. For this purpose we augmented the open-source PREVAIL eBPF verifier [48] with verification for Janus helper functions and output schemas. The controller further instruments the verified bytecode with additional control code that pre-empts it if its runtime exceeds some threshold (see §5.1). The (patched) codelets are JIT compiled and pushed to Janus devices over the network, along with metadata files required for enabling the flexible output of data using protobuf output schemas (see §5.2). The controller provides a data collector, which collects and deserializes the data sent from the Janus codelets. Janus also provides an SDK that allows developers to locally test Janus codelets early in the development cycle. The SDK includes a compiler, a verifier and a debugger, as well as the definitions of all the helper functions and map types that are supported by Janus devices.

## 3.2 New vRAN RIC capabilities

We describe the new monitoring and control capabilities that Janus enables, through a simple, yet realistic, example (Listing 1). The example refers to a Janus codelet developed for the vDU of OpenAirInterface [26]. The codelet is invoked by a hook that is introduced at the FAPI interface ([88], Fig 1). FAPI messages are C structures with information about the scheduling of radio resources to UEs. In this codelet, a counter maintaining the number of captured FAPI messages is sent to the data collector once every 1000 events. While simple, this codelet captures important features that demonstrate the power of Janus over the conventional RIC design.

```
1  struct janus_load_map_def SEC("maps") countermap = {
2    .type = JANUS_MAP_TYPE_ARRAY,
3    .key_size = sizeof(uint32_t),
4    .value_size = sizeof(uint32_t),
5    .max_entries = 1,
6  };
7
8  struct janus_load_map_def SEC("maps") outmap = {
9    .type = JANUS_MAP_TYPE_RINGBUF,
10   .max_entries = 1024,
11   .proto_msg_name = "output_msg",
12   .proto_name = "output_msg",
13   .proto_hash = PROTO_OUTPUT_MSG_HASH,
14 };
15
16 SEC("janus_ran_fapi")
17 uint64_t bpf_prog(void *state) {
18   void *c;
19   uint32_t index = 0, counter;
20   nfapi_dl_config_request_pdu_t *p, *pend;
21   output_msg s;
22
23   struct janus_ran_fapi_ctx *ctx = state;
24   p = (nfapi_dl_config_request_pdu_t *)ctx->data;
25   pend = (nfapi_dl_config_request_pdu_t *)ctx->data_end;
26
27   if (p + 1 > pend) return 1;
28
29   if (p->ndlsch_pdu > 0) {
30     c  = janus_map_lookup_elem(&countermap, &index);
31     if (!c) return 1;
32     counter = (*(int *)c + p->ndlsch_pdu);
33     if (counter == 1000) {
34       s.counter = counter;
35       janus_ringbuf_output(&outmap, &s, sizeof(s));
36       counter = 0;
37     }
38   }
39   return 0;
40 }
```

**Listing 1: Example Janus codelet**

**Secure access to rich vRAN data:** The `state` argument in line 17 of Listing 1 is the context passed to the Janus hook that contains a pointer to a vRAN FAPI structure [41, 81] (line 24). It describes the scheduling allocation for a particular downlink slot, comprised of more than 20 fields per user, including modulation and coding scheme, transport block size, allocated resource blocks, MIMO etc. The verifier ensures read-only access to the internal vRAN context information. Due to the modular vRAN design, there is a small number of similar interfaces specified in different standards (3GPP, Small cell forum) that carry all relevant state across vRAN components. By adding hook points at these interfaces we can give Janus developers access to a large trove of vRAN telemetry. We have identified and implemented these hooks (Table 1) and we demonstrate in §4 how they can be used to enable multiple RIC applications without modifying a single line of code inside the vRAN functions. Note that the implementation of an interface may differ across vRAN vendors (e.g., in terms of C struct memory layout). Due to the flexibility of Janus, we were able

| Hook point | Actuator helper function | Type of control |
|---|---|---|
| Inter-slice radio resource scheduling | `allocate_slice_rbs()` | Distribute radio resource blocks across slices |
| Intra-slice radio resource scheduling | `allocate_ue_rbs()` | Make scheduling decisions for UEs within a slice |
| Uplink power control | `set_uplink_ue_power()` | Control uplink power of UEs |
| Link adaptation | `set_ue_mcs()` | Adjust the maximum modulation & coding scheme for scheduled UEs |

**Table 2: Janus control hooks introduced in commercial-grade vDU network functions and OpenAirInterface.**

to adjust to the observed differences while maintaining the same codelet functionality with minor codelet changes.

**Statefulness:** Janus codelets rely on shared memory regions known as *maps* to store state across consecutive invocations and to exchange state with other codelets (c.f. §4 and appendix A.1). Janus provides various map types for storing data, including arrays, hashmaps and Bloom Filters. In this example, we maintain a counter of FAPI packets using a single-element array map (lines 1-6). On each invocation, the counter reference is restored from memory through a helper function (line 30) and incremented with the new number of packets (line 32). Various safety checks are required to enable static verification (e.g. lines 27 and 31).

**Flexible output schemas:** Janus codelets can send arbitrary telemetry data to the data collector using flexible output schemas through a special type of ringbuffer map (lines 8-14). This map is linked to a codelet-specific protobuf schema defined by the codelet developer (see §5.2). This example uses a custom protobuf schema called `output_msg` (line 21), with a single counter field (line 34). The data is exported to the data collector through a helper function (line 35). This flexibility allowed us to implement the data models currently available in the O-RAN RIC specs *without modifying a single line of code in the vRAN*, after our Janus vRAN hooks were in place.

**Safe & expressive custom control operations:** Janus codelets cannot directly modify the state of the vRAN. Instead, modification of the vRAN behavior is done through *actuator helper functions*, provided by the vRAN vendors, which are responsible to apply the changes decided by the control codelet. Given the small number of real-time control loops in the vRAN functions that only modify standard compliant 3GPP parameters (present in the implementations of all RAN vendors), we believe that the use of actuator helper functions is reasonable. Based on this, we identified and implemented a number of control hooks, listed in Table 2, which, we believe, capture most of the critical tight control loops that can be used for novel control applications. As an example, we used the *inter-slice radio resource scheduling* hook point to implement 3 network slicing algorithms from the literature in OAI (see §4), by modifying the number of resource blocks allocated to each slice using the `allocate_slice_rbs()` actuator helper function.

## 4 NOVEL JANUS USE CASES

Here, we illustrate the values of Janus using several representative examples of telemetry, inference and control applications that we built (for evaluation see §7).

| Name | Type | LOC | Short | Ref |
|---|---|---|---|---|
| Total DL PRB Usage | M | 207 | KPM1 | 5.1.1.2.1 [13] |
| Total UL PRB Usage | M | 171 | KPM2 | 5.1.1.2.2 [13] |
| Distr. of DL PRB Usage | M | 232 | KPM3 | 5.1.1.2.3 [13] |
| Distr. of UL PRB Usage | M | 197 | KPM4 | 5.1.1.2.4 [13] |
| Total num. of initial DL TBs | M | 152 | KPM5 | 5.1.1.7.1 [13] |
| Total num. of DL TBs | M | 156 | KPM6 | 5.1.1.7.3 [13] |
| Total num. of initial UL TBs | M | 155 | KPM7 | 5.1.1.7.6 [13] |
| Total num. of UL TBs | M | 157 | KPM8 | 5.1.1.7.8 [13] |
| Raw DL scheduling info | M | 137 | RAW1 | - |
| Raw UL scheduling info | M | 150 | RAW2 | - |
| Interference detection | I | 265 | ID | - |
| ARIMA | I | 81 | ML1 | [33] |
| Decision tree | I | 10495 | ML2 | [45] |
| Random forest | I | 51 | ML3 | [98] |
| Earliest deadline first slicing | C | 174 | SL1 | [52] |
| Static slicing | C | 41 | SL2 | [74] |
| Proportional fair slicing | C | 384 | SL3 | [74] |

**Table 3: Monitoring (M), control (C) and inference (I) Janus use cases we developed, with lines of code (LOC).**

**Flexible monitoring:** We use Janus to implement codelets that allow us to extract KPIs specified in the KPM model of O-RAN [21] (lines 1-8 in Table 3), as well as raw scheduling data (lines 9-10) *without having to change a single line of code in the vRAN functions*. This demonstrates the ability of Janus to build new and change existing O-RAN service models [31, 47] on the fly, without undergoing a lenghty standardization process. For example, we were able to collect the downlink total Physical Resource Block (PRB) usage KPI [13], by tapping into our Janus FAPI hook of Table 1 and capturing the `nfapi_dl_config_request_pdu_t` struct that was described in Listing 1 and which contains the number of PRBs allocated to each user at each scheduling decision. The contents of this struct were stored in a Janus map, averaged over a period of 0.5ms and sent out to the Janus data collector.

**Low overhead, real-time inference:** To demonstrate how Janus can overcome the data volume limitation, described in §2.2, we developed a codeletset that detects external radio interference by transforming an already operational 5G radio unit into a spectrum sensor. Our codeletset is composed of two codelets that use maps for coordination. The first detects idle slots when there are no 5G transmissions (installed at the FAPI hook of Table 1). The second samples interference during the detected idle slots (installed at the IQ samples hook of Table 1). The flexibility of Janus allows us to adjust the fidelity and overhead of the interference detector as needed, by specifying a number of parameters in terms of which antenna ports and symbols to collect IQ samples from, with what frequency

(e.g., every reception slot or every 10ms) and granularity (e.g., raw IQ samples vs average energy per resource block). As we show in §7.2, performing this inference inline in the vRAN, instead of exporting raw IQ samples to the RIC, allowed us to *reduce the telemetry bandwidth by a factor of 40*. A similar approach can be used to implement other inference use cases that require radio channel telemetry data in a different format (e.g., wireless localization [62, 86] and channel estimation [37, 38]).

Furthermore, many RAN control loops require real-time parameter prediction, since the prediction freshness has a direct impact on the network performance [28, 29, 40, 46, 66, 83, 89, 98, 100]. Due to the O-RAN RIC latency, xApps provide predictions that are 10s of milliseconds old, while real-time inference is currently not supported [23, 77]. Using Janus, we were able to build codelets that perform inference inside the vRAN functions *with prediction latency under* 10*ms*. We demonstrate this functionality by building the inference models listed in lines 12-14 of Table 3. The first is an ARIMA time-series model for the prediction of user signal quality, following a methodology similar to [33]. The second is a quantile decision tree for the prediction of signal processing task runtimes, using the methodology in [45]. More complex models, such as the Random Forest in [98], are more difficult to implement in Janus C code, as they result in a large number of bytecode instructions ($>$ 100K) making the verification process slow ($>$ 20 minutes). To overcome this, we added Janus support for Random Forests in the form of a map (`JANUS_MAP_TYPE_ML_MODEL`). A pre-trained serialized random forest model can be passed to Janus and linked to this map during the codelet loading. Janus parses the serialized model to verify it and reconstructs it in memory. The model can then be accessed by the codelet for inference using a helper function `janus_model_predict()`. This approach is similar to the serialization feature offered by frameworks like Tensorflow for micro-controllers [91] and could be extended to other commonly used ML models (e.g., LSTM). For more details about this process, please see appendix A.4.

**Real-time control:** Many enterprise applications require network slicing for stricter service QoS guarantees [39, 44] . Existing O-RAN service models allow for a set of pre-defined slice scheduling policies [34, 60, 85] that control scheduling at 10s of milliseconds granularity. Using Janus, we *enabled real-time slicing with arbitrary scheduling policies with a granularity of 0.5-10ms*, something impossible with today's RIC model. We relied on the inter-slice radio resource scheduling hook of Janus from Table 2, which is invoked by the MAC scheduler in the beginning of each scheduling period. The hook receives the scheduling state of the base station as context (number of devices, the slice they belong to, buffer sizes, signal quality etc). Using this hook, we were able to implement three network slicing schedulers as Janus codelets, as listed in Table 3 (lines 15-17). All three schedulers apply their scheduling decision to the base station using the `allocate_slice_rbs()` actuator helper function of Table 2.

# 5 SYSTEM DESIGN CHALLENGES

## 5.1 Runtime control

The existing eBPF verifier can assert memory safety and termination – if a codelet does not provably terminate (e.g. due to unbounded loops), it is rejected. However, as explained in §2.3, it does not give sufficiently tight guarantees on the codelet worst-case execution time.

*5.1.1 Challenge of estimating runtimes.* One simple approach to estimating the worst-case execution time is to analyze the maximum number of eBPF instructions a codelet can execute. This information is inferred through static analysis for the codelet's longest path, taking into account bounded loops. However, it is very difficult to translate the number of instructions into the expected runtime, as this can depend on a number of factors, including the CPU clock, the memory and cache hierarchy, the translation of the eBPF instructions to JIT code etc. [36, 94]. An additional challenge for Janus are the helper functions, whose execution time can widely vary between functions and across parameter values.

To illustrate these challenges, consider the codelets in Listings 2 and 3. Both perform a 1000 iterations loop, with the first calling a helper function inside the loop. The verifier indicates that the codelet of Listing 3 requires 64 more instructions compared to the one of Listing 2. However, for a reference Xeon Platinum 8168 CPU @ 2.7GHz, we observe that the codelet of Listing 2 is more expensive (runtime of 4.3 $\mu$s vs 2.4 $\mu$s for the codelet of Listing 3). This is because the helper function incurs a more significant overhead compared to the multiplication and addition instructions of the other codelet, indicating that the maximum number of instructions per codelet is not a good proxy of the max runtime.

```
1 for (int k = 0; k < 1000; k++) {
2   index = 0;
3   c = janus_map_lookup_elem(&counter, &index);
4   s.counter = k + 10;
5 }
```

**Listing 2: Loop w/ helper function (avg runtime: 4.3 $\mu$s)**

```
1 for (volatile int k = 0; k < 1000; k++) {
2   counter += i;
3   counter2 = counter * i;
4   s.counter += counter2;
5   i++;
6 }
```

**Listing 3: Loop w/o helper function (avg runtime: 2.4 $\mu$s)**

*5.1.2 Enforcing runtime through bytecode patching.* To address these challenges, Janus injects instructions in the eBPF bytecode that measure the codelet execution time while running and preempts it if a threshold is exceeded. As illustrated in Fig 3, a helper function (`mark_init_time()`) is added at the beginning of the codelet, which stores the current time

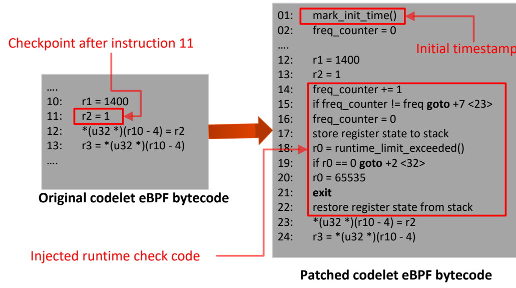



**Figure 3: Simplified version of code patching process.**

(using Intel's `rdtsc` instruction) in a thread local variable. The patcher introduces checkpoints in selected locations that invoke a helper function (`runtime_limit_exceeded()`, line 18 in Fig 3), which checks the elapsed runtime of the codelet since `mark_init_time()` and compares it against a threshold. If the threshold is exceeded, the codelet is forced to exit and return an error (lines 19-21 in Fig 3). The runtime threshold is specified per codelet during the loading of the codeletset (see appendix A.1 and Listing 6). Finally, the patcher updates all jump offsets to account for the injected instructions. This approach allows us to verify the patched bytecode for safety, ensuring that any modifications made by the patcher do not affect the safety of the codeletset.

The time check is implemented as a helper function, because it calls the Intel `rdtsc` instruction, which does not have a counterpart in the eBPF instruction set. The helper function call invalidates eBPF registers `r0` - `r5`, which could be storing state from the normal codelet execution flow. To ensure verifiability, Janus stores and reloads the values of those registers (lines 17 and 22 in Fig 3). This requires that codelets have at least 48 bytes free in their stack (eBPF functions have 512 bytes stacks). We believe that this is a reasonable requirement, given that codelets can always use maps to store more state.

**Points of injection:** A key question when patching is where to inject the checkpoints. We want to limit the maximum number of instructions $N$ between two consecutive checkpoints to reduce the effect of the runtime jitter (shown in Listings 2 - 3). However, each checkpoint incurs overhead (a call to the helper function `runtime_limit_exceeded()`, saving and restoring registers, etc.). All this adds up to more than ~24ns per checkpoint for a reference Xeon Platinum 8168 CPU @ 2.7GHz. To keep the overhead low, Janus spreads the checkpoints using the greedy Algorithm 1. Initially, Janus adds checkpoints right after the invocation of helper functions marked by the vendors as *long lasting* and thus potentially unsafe, if used often (line 2). Next, it uses the static analysis of the verifier to enumerate (from shortest to longest) all the simple paths from the first instruction of the codelet to the last, as well as all the cycles. For each path, Janus adds a checkpoint every $N$ instructions (lines 11-14). The algorithm takes into account checkpoints that have already been added during the traversal of other paths. If a checkpoint is found, the counting of instructions is reset, using the existing checkpoint as the starting point (lines 8-9). At least one checkpoint is added on each cycle even if the distance is smaller than $N$ (lines 17-19). This guarantees that a checkpoint can always be reached once every $N$ instructions.

**Algorithm 1:** Checkpoint injection decision

**Data:** $N > 0$, list $F$ of codelet instructions, where long lasting helper functions are called, ordered list $P$ of all simple codelet paths from first to last instruction and cycles (increasing length)
**Result:** List $C$ of checkpoint instructions positions

```
1  C ← {};
2  foreach instruction f in F do C ← C + f;
3  foreach p in P do
4      ins ← 0;
5      fins ← first instruction of p;
6      foreach instruction i in p do
7          ins ← ins + 1;
8          if i has already checkpoint then
9              ins ← 0;
10         else
11             if ins = N then
12                 C ← C + i;
13                 ins ← 0;
14             end
15         end
16     end
17     if p is cycle and no checkpoint was added then
18         C ← C + fins;
19     end
20 end
```

For finer control, Janus allows vendors to instrument checkpoints in their long lasting helper functions using a macro (`RUNTIME_LIMIT_EXCEEDED`), which performs a similar operation as the patch of Fig. 3. For example, in the case of the random forest model discussed in §4, we added such checks after the inference of each estimator (tree) of the model.

Even for a few checkpoints, the overhead can become significant (e.g., in codelets with tight loops). To further reduce the overhead, the patch code performs checks with a sampling frequency (1 out of $M$ checkpoint hits). The patcher adds a 32-bit counter in the eBPF stack and performs a check only when this counter reaches some value (line 15 of Fig 3, right); otherwise, the execution flow jumps back to the original bytecode instruction, This way, a check is guaranteed to be performed at least once every $M \times N$ instructions.

**Pre-empted control loops:** Each control hook must provide a default control decision, if a loaded control codelet is pre-empted and does not come up with one. A pre-empted codelet returns a `CONTROL_FAILED` code and the default action provided by the RAN vendor is executed (as shown in Listing 4). For example, in the case of the inter-slice radio resource scheduling hook of Table 2, the default action of the vendor could be to assign the radio resource blocks equally among the slices. Similarly, for the link adaptation hook, it could

be to set a robust modulation and coding scheme, that will provide low error rate regardless of the signal conditions of the attached UEs. Finally, Janus provides a helper function `check_preemption()`, which allows a codelet to check if it was pre-empted during its previous run. This allows codelets to reset their operation, if they have dirty state.

```
decision = hook_custom_janus_control_operation(ctx);
if (decision != CONTROL_SUCCESS)
  decision = call_default_control_operation(ctx);
```

**Listing 4: Pseudo-code for Janus custom control hook**

## 5.2 Flexible and verifiable output schemas

Janus allows the definition of arbitrary user-defined output schemas, loaded with the codelet at runtime. The output data is transmitted over a network to a centralized controller (more details in appendix A.2). Janus serializes the data using protobufs. However, adding arbitrary schemas can compromise safety. Specifically Janus has to deal with two challenges.

```
message Example {
   repeated int32 element = 1;
}
Example.element max_count:16
```

**Listing 5: Example of output schema definition with variable size fields (up to 16 elements).**

The first is making sure that codelets cannot generate arbitrarily large serializable messages, which could violate memory safety. Protobuf messages are defined by the developer and can contain variable size fields (e.g. repeated fields or strings). The Janus verifier is not aware of the actual size of a message at compile time, hence it cannot statically verify it. To overcome this problem, Janus requires an upper bound for all protobuf schemas with variable-sized fields in the message specification, as shown in Listing 5. Janus allocates the maximum message size for the C representation exposed to the codelet, and reports the size to the verifier (in this case 16× `sizeof(int32) + sizeof(int16)` = 66B). This allows for static verification at the expense of slightly increased memory consumption (which is not a bottleneck in a vRAN system).

The second challenge is making sure that an incorrectly formatted message cannot lead to memory violations. Consider a case where a programmer allocates a 30B memory chunk, casts it as an `Example` message, sets the number of elements to be 16 and calls the protobuf encoder. Since the memory chunk is too small for 16 elements, the encoder will attempt to encode from a memory outside the allocated chunk, which may lead to a segfault. To ensure memory access safety, we modify the verifier to assert that the memory passed to the protobuf encoder is always equal to the maximum possible size (66B in this case). Bugs like the one above will still be functionally incorrect and send garbled data, but do not violate safety.

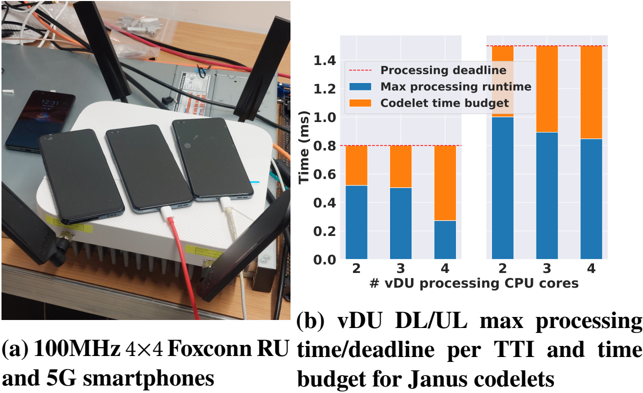
# 6 IMPLEMENTATION & INTEGRATION

Here, we provide more details about the implementation of the Janus components and their integration with a real vRAN.
**Janus device:** The Janus device is implemented in C as a library that can be dynamically linked to vRAN functions. It is based on a usermode implementation of eBPF [59], which we extended to add support for the Janus maps, helper functions and the mechanism for exporting output data. Overall, we had to add ∼5K lines of code to the basic implementation. The Janus device code was developed without making any assumptions about the threading model of the vRAN functions (e.g., affinity of the threads that call hooks, their scheduling policy and priority etc.). However, Janus can be configured to allow the further optimization of the library if such information is known. We have taken great care in ensuring that the fast-path of Janus (where hooks might be invoked in time-critical parts of the vRAN functions), will never be blocked or pre-empted. You can find more details about our real-time optimizations in appendix A.5.
**Janus controller:** The Janus controller is written in Go (data collector) and Python (codelet loader/patcher), with ∼4K lines of code. The controller communicates with Janus devices through a TCP-based API using protobuf. This could be replaced by other interfaces, like O-RAN RIC E2 interface. For the codelet verification, we used the open source PREVAIL verifier [48], which we extended with ∼1K lines of code to add support for Janus specific functionalities (i.e., helper functions, maps etc). Finally, the Janus patcher relies on pyelftools [1] and LLVM [8] to manipulate the codelets' ELF file contents.
**Janus SDK:** The Janus SDK is written in Python (∼1K lines of code) and shares parts of its codebase with the Janus controller. It relies on LLVM for the compilation of codelets to eBPF bytecode, on eBPF [59] for the conversion of the bytecode to x86 JIT code and on nanopb [6] for the compilation of protobuf messages for the codelets' output schemas.
**vRAN integration:** Integrating Janus to vRAN functions is simple and fast. To demonstrate this, we integrated Janus devices to two vRAN software implementations as a proof-of-concept. One is the commercial-grade 5G vCU/vDU implementation developed by CapGemini [35], and based on Intel's FlexRAN PHY design [57]. The other is the open source OAI [26]. Both are written in C/C++ and the integration and linking of Janus code was straightforward. For the integration of Janus we had to add approximately 50 lines of initialization code in each vRAN function that we tested, as well as ∼30 lines of code for each new hook we introduced.

# 7 PERFORMANCE EVALUATION

## 7.1 Experimental setup

**Hardware and software setup:** For the evaluation of Janus we use a server equipped with 48 physical cores (Intel Xeon




**(a) 100MHz 4×4 Foxconn RU and 5G smartphones**

**(b) vDU DL/UL max processing time/deadline per TTI and time budget for Janus codelets**

**Figure 4: vRAN testbed infrastructure and performance**

Platinum 8168 @ 2.7GHz) and 196GB of RAM with hyper-threading disabled. The server is running Linux v5.15 with the `PREEMPT_RT` real-time patches applied [82] and optimizations for real-time performance, including disabled P and C-States and hugepages of 1GB. We opted for this configuration, as it is typical for the deployment of vRAN functions [45, 58, 93].

For the evaluation, we use three setups. The first is an end-to-end setup, composed of the commercial grade 5G vRAN stack from CapGemini [35], with integrated Janus devices (see §6), a commercial-grade 5G core, a 100MHz $4 \times 4$ Foxconn radio unit and 5G OnePlus Nord smartphones (Fig 4a). Our vRAN is x86-based and, except for LDPC [55], all tasks run on x86 processors. Using this setup, we are able to generate a max of 1Gbps downlink and 45Mbps uplink traffic. This setup is instrumented with Janus collecting IQ samples, FAPI and RLC data (see Table 1). In the second setup we instrument 4G OAI with hooks to RRC/F1 and FAPI data, as well as the inter-slice radio resource allocation hook of Table 2. The final setup provides a dummy Janus device, which is a single-threaded process that runs in a loop and invokes codelets attached to Janus hooks. We use this setup for microbenchmarks (§7.3).

In all setups, the Janus hooks are being invoked by threads that have their affinity set to a single core and are scheduled using the `SCHED_FIFO` policy (non-preemptible), with a scheduling priority of 94. For all our runtime measurements, we use a time measuring framework based on the guidelines in [76].

**Codelet time budgets in vRAN:** We use our 5G RAN setup to determine how much time we can allocate to Janus codelets without affecting the RAN performance. We focus on the PHY layer of the vRAN DU, as all other layers have less stringent timing requirements. Transmissions and receptions of packets in the PHY occur in Transmission Time Intervals (TTIs) of a fixed duration [45]. Using our CapGemini vRAN setup we measure the runtimes of the PHY per TTI when fully saturated with 4x4 MIMO traffic (1Gbps DL and 45Mbps UL) over a period of 15 minutes. In Fig 4b we show the maximum runtimes for uplink and a downlink as a function of the number of required CPU cores. We also plot the processing deadline for the given configuration based on the vendor guidelines. The difference (orange), is the maximum runtime budget for Janus codelets, and it varies from 200 $\mu$s to 600 $\mu$s.

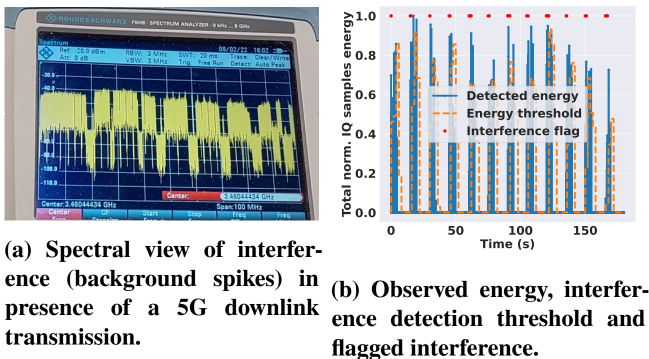
While these numbers may seem high, in practice the limits are much smaller for several reasons. Some codelets may be executed multiple times per TTI. For example, the IQ sample processing from §4 is called 14 times per TTI (one per OFDM symbol). Furthermore, multiple codelets can be loaded on different hooks, sharing the overall time budget. Finally, more demanding PHY configurations, such as massive MIMO (which we are not able to evaluate at the moment), will likely leave less spare CPU time for Janus hooks. Our design goal assumes a codelet run-time budget can be as low as 20$\mu$s.

### 7.2 End-to-end system evaluation

Here we explore the behavior and benefits of Janus in a real end-to-end deployment, by demonstrating the flexibility of extracting telemetry data using Janus codeletsets and the safety provided by Janus when loading the codelets. For the safety part, we focus our evaluation on the runtime control mechanism described in §5.1. The effectiveness of the static analysis of the eBPF verifier that is used by the Janus controller is extensively studied in [48]. Similarly, we point the reader to the references of Table 3 for the performance results of the algorithms we used for our implemented codelets.

We use the interference detection application described in §4 as a representative example codeletset for our analysis. We deploy a USRP software-defined radio as an external interferer that generates a repetitive interference pattern with 5s of interference and 5s of silence. A spectrum view of the interference is shown in Fig 5a (thin spikes) during a real 5G downlink transmission. We run downlink iperf measurements between one of the OnePlus Nord 5G phones and the 5G vRAN and we see about 30% of packet loss when the interferer is active. While running the measurement, we load the Janus codelets for interference detection, described in §4. We implement a simple interference detector at the controller that continuously tracks the mean and the variance of the input signals per resource block and declares interference if the input is larger than the mean plus 3 times standard deviation. We show in Fig. 5b that this approach successfully detects all interfering periods.

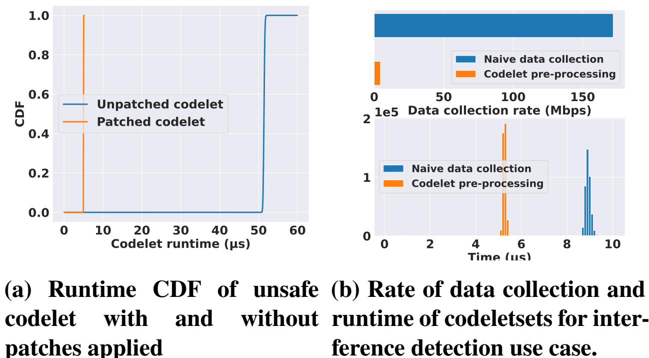
**Codelet patching prevents RAN crashes:** Next, we show how Janus can effectively deal with codelets that, while verifiable, can be unsafe for the operation of the vRAN due to long execution times. We wrote a different codelet for the same experiment that is correct, but deliberately written to be inefficient. It allocates 13KBs of memory for a temporary struct, memsets the memory with zeroes byte by byte in a tight for loop and then copies the IQ samples passed by the hook one by one in a second for loop before sending them to the controller.




**(a) Spectral view of interference (background spikes) in presence of a 5G downlink transmission.**

**(b) Observed energy, interference detection threshold and flagged interference.**

**Figure 5: Spectral view of interference and detection using data collected through Janus.**




**(a) Runtime CDF of unsafe codelet with and without patches applied**

**(b) Rate of data collection and runtime of codeletsets for interference detection use case.**

**Figure 6: Pre-emption of unsafe codelet and benefits of programmability with codelets for data collection**

Our deliberately inefficient codelet is verified as correct by the verifier, as it is deemed safe in terms of memory access, as well as provably terminates (bounded loop). However, once this codelet is loaded to our end-to-end vRAN deployment unpatched, it crashes the vRAN. As it can be seen by the CDF in Fig 6a, this codelet runs for 51.4 $\mu$s on the median and 52.2 $\mu$s on the 99.999 percentile. Given that the hook of the raw IQ samples is called 14 times for each TTI (one per OFDM symbol), the codelet runs for a total of 719.6 $\mu$s on average, which is greater than the 600 $\mu$s time budget that we have for the UL chain of the vRAN (shown in Fig 4b).

We then patch the codelet using the Janus patcher and we re-load it to the vRAN with a runtime threshold of 5 $\mu$s. As it can be seen in Fig 6a, the patched version of the codelet is pre-empted early and so the median runtime now becomes exactly 5 $\mu$s and the 99.999 percentile becomes 5.04 $\mu$s. While this codelet no longer sends IQ samples out (as it is pre-empted), the vRAN remains protected, as no deadlines are violated.

**Reduction in data collection bandwidth:** In the same interference detection example, the two codelets coordinate their outputs to reduce the overall data collection bandwidth, as explained in §4. In order to evaluate the benefit of coordination, we also implement the same scenario using a more centralized approach where the coordination happens at a RIC. In this approach, two codelets independently send all of the scheduling data and the raw IQ samples to the controller (~ 13KB of data per symbol) instead of correlating their inputs locally and sending only IQ samples for the idle slots.

For these setups, we measure the throughput for sending the output data to the Janus collector, as well as the runtime of the two codeletsets, as is illustrated in Fig 6b. As it can be seen at the top sub-plot of the figure, the naive interference detection method results in a data collection rate of 172Mbps, while the pre-processing method inside the vRAN results in a collection rate of only 4.5Mbps, almost 40× less. The runtime of the codelets with pre-processing is lower by ~ 3.5 $\mu$s compared to the naive case, despite the extra pre-processing work. This is because the naive approach requires a memset and a memory copy of 13KBs each time the codeleset sends the raw IQ samples out, while the pre-processing approach only requires ~400B per call. Custom pre-processing is impossible with the O-RAN RIC, since a service model has been specified for the use case and integrated by the RAN vendors.

**Codelets runtimes:** Due to lack of space we don't discuss each scenario from §4 in detail. Instead, we report the median and tail (99.999) runtimes of the 17 codelets of Table 3 in Fig 7, using the shorthand names of the table as a reference. We consider the worst case execution case for each codelet (i.e., maximum number of devices, maximum bandwidth etc). The reported times are for patched codelets. We use a patching distance $N = 60$ for checkpoints and a sampling frequency of $M = 10$ (see §7.3 for details on the choice of parameters). We observe that the runtime of all the codelets is well below the 20$\mu$s time budget discussed in §7.1 (< 8$\mu$s for the worst codelet), with the most demanding being the slicing schedulers (SL1 and SL3), the interference detection codeletset (ID) and the raw scheduling data monitoring codelets (RAW1 and RAW2). We further demonstrate this by deploying all the codelets marked as monitoring in Table 3 on our 5G vRAN deployment at the same time, while saturating the network with traffic (1Gbps DL and 45Mbps UL). We do not observe any change in the link performance after loading the codelets.

## 7.3 Microbenchmarks

**Patching overhead and reactiveness:** Here, we explore the behavior of the patching process (§5.1), by studying the most computationally demanding codelets of Table 3, based on the runtime results of §7.2 (i.e., `RAW1`, `ID`, `SL1` and `SL3`). The remaining codelets of Table 3 present similar patching behaviors and thus are omitted, due to space constraints.

First, we study Algorithm 1 in terms of the number of introduced checkpoints for various checkpoint distances $N$. As we can see in Fig 8a, the more instructions a codelet has

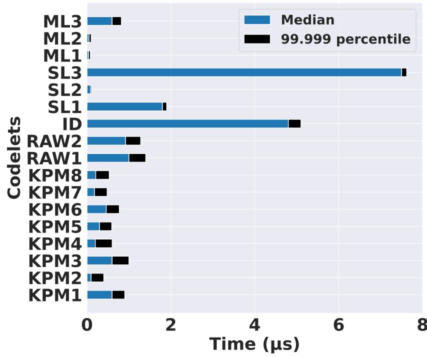



**Figure 7: Patched codelets (Table 3) worst-case runtime.**

(listed under the label of each codelet), the more checkpoints are introduced. Moreover, as we increase $N$, the number of checkpoints drops. The number of checkpoints is in almost all cases slightly higher than the number of instructions divided by $N$, meaning that some checkpoints have a distance smaller than $N$, if the code was to be executed sequentially. Inspection of the bytecode reveals that the excess checkpoints are mainly introduced in tight loops ($< N$ instructions), which, if unrolled, form a block of more than $N$ instructions, demonstrating that our patching algorithm can effectively capture such cases.

Next, we study the behavior of codelets for various patching distances (parameter $N$) and sampling frequencies (parameter $M$). The results in Fig 8b show the runtime of patched codelets without a runtime threshold, compared to the unpatched version. The runtime overhead can become significantly high for a small $N$ (e.g. more than 100% for `ID`), because runtime checks are executed very often, while reducing the sampling frequency can help (e.g., as shown in the case of $N = 10$ and $M = 30$ for `ID`). On the other hand, a large value of $N$ and a reduced sampling frequency (large $M$) is translated to less runtime checks, leading to higher mean and tail latencies for pre-empting codelets, as shown in Fig 8c (runtime threshold set to 400ns). For $N = 60$ and $M = 30$, the tail runtime of `RAW1` and `ID` is almost 100% more than the runtime threshold. Based on our evaluation of all the codelets of Table 3, we find that the values $N = 60$ and $M = 10$ draw the best balance between runtime overhead and pre-emption latency.

Finally, we compare the checkpoint method of Algorithm 1, with an alternative method proposed in [95, 96], where checkpoints are introduced on each basic block of the control flow graph of the patched code. For the basic blocks method, we use a sampling frequency of $M = 30$, which yields similar pre-emption tail latency results to the Janus patcher for $N = 60$ and $M = 10$. As shown in Fig. 8d, the basic blocks approach incurs in most cases higher runtime overhead (99.999 tail) compared to the Janus patcher (e.g., more than 2× higher overhead for the `SL1` codelet). The reason for this increased overhead is that in many cases, basic blocks can be very small (2-3 instructions). If such a basic block is visited often (hot code), then the instructions added by the checkpoint can more than double its runtime, even if sampling frequency checks are used. The approach taken by Janus is more disciplined in the sense that it allows the RAN operator or vendor to choose the exact number of instructions between two checkpoints.

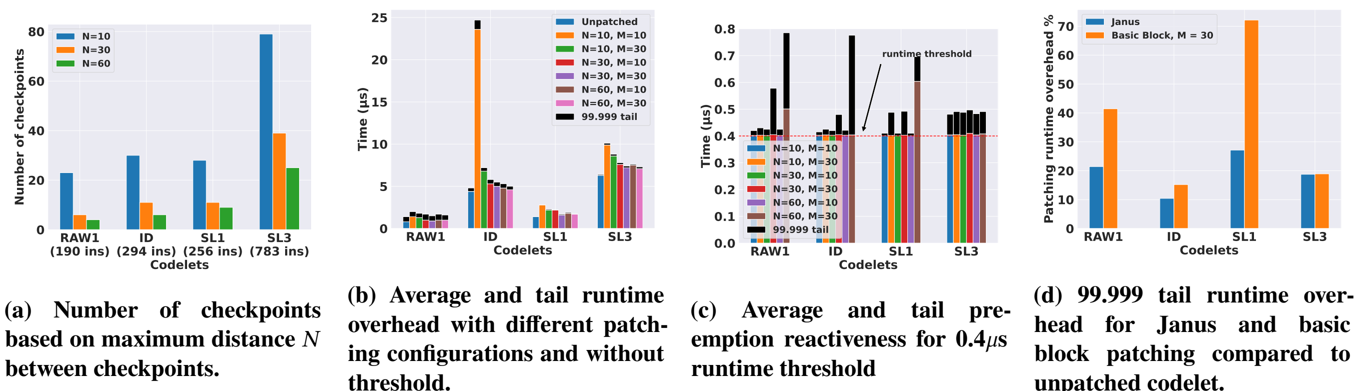
| | Median | 99.9 | 99.999 |
|---|---|---|---|
| Single empty janus hook | < 1ns | < 1ns | < 1ns |
| 10 empty janus hooks | < 1ns | 10ns | 10ns |

**Table 4: Median/tail execution time of empty Janus hook.**

**Janus hook overhead:** To measure the overhead of idle Janus hooks, we use the dummy Janus device and measure the elapsed time for calling a single or 10 Janus hooks, without any codelet loaded, over 2M iterations. The results are presented in Table 4 for the median, 99.9 and 99.999 percentile. As we can observe, in the case of a single hook call, the overhead is negligible (< 1ns) for all cases. The difference goes up to 10ns for the 99.9 and 99.999 percentile in the case of the 10 hooks (∼ 1ns per hook call). We conclude that adding hooks to the vRAN code has negligible impact on its performance.

**Codelet overhead when extracting data:** We next evaluate the codelet overhead when copying data to the output messages and placing it into the output map (this does not include the overhead of the output thread and protobuf serialization). We base our benchmarking on a protobuf message `SizeMessage1` defined as part of the benchmarking suite of the protobuf library [79]. This message contains 62 fields in total, both simple (e.g., `int32`, `int64`, `bool`) and variable sized (`string` and repeated fields). We write a codelet that populates a `SizeMessage1` with random content and for different message sizes, and sends it. As we can observe in Fig 9a, for small packet sizes (< 2KB) the execution runtime both at the median and the tail remains below 1 $\mu$s and then gradually increases for large packet sizes, but always remains below 2$\mu$s even for jumbo packets of 9KB. In practice, none of the codelets that we wrote for the use cases in §4 required to send monitoring packets of more than 2KB, meaning that the output overhead for most practical scenarios is very low.

**Networking overhead:** Finally, we measure the overhead of the output thread that serializes protobuf messages and sends them over the network to the controller. We use the same setup as in the previous experiment and we measure the maximum achievable packet sending rate. The results are illustrated with the blue line in Fig 9b. For small packets ($< 500B$), Janus can serialize and send more than 80kpps in a single output thread, which drops to 20kpps when sending jumbo packets of 9KB. Based on the telemetry codelets (§4), the number and size of packets for a single cell, even in the most demanding use cases, falls within the orange area of Fig 9b. This means that a



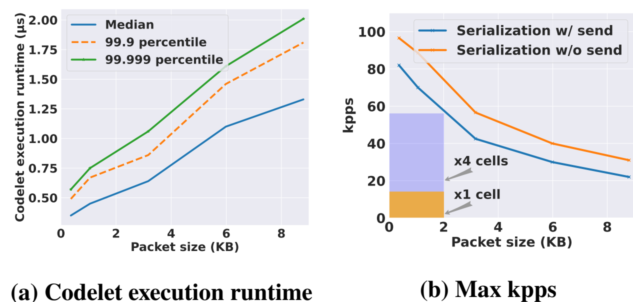

**(a) Number of checkpoints based on maximum distance $N$ between checkpoints.**

**(b) Average and tail runtime overhead with different patching configurations and without threshold.**

**(c) Average and tail pre-emption reactiveness for 0.4$\mu s$ runtime threshold**

**(d) 99.999 tail runtime overhead for Janus and basic block patching compared to unpatched codelet.**

**Figure 8: Analysis of codelet patching behavior and comparison with alternative approaches.**




**(a) Codelet execution runtime**

**(b) Max kpps**

**Figure 9: Codelete execution runtime and maximum kpps for `SizeMessage1` message [79] with fields of varying size.**

single Janus device can handle up to 4 cells (light blue and orange area) for demanding use cases and more than double for lightweight monitoring. Currently, the main bottleneck is the serialization of the output messages. This can be seen with the orange line of Fig 9b, since even when we drop packets instead of sending them, we only see a 25% increase in the output rate. This could be further reduced by replacing socket-based UDP operations with kernel bypassing (e.g. DPDK).

## 8 DISCUSSION AND RELATED WORK

**Pushing arbitrary code to vRAN functions -** The works in [43, 73] argue about the need for real-time RAN programmability, by loading arbitrary code in the vRAN functions at runtime. While these works are conceptually similar to Janus, they don't propose a safe way to implement such features and acknowledge that the safety concerns make their proposal unacceptable for realistic deployments. They are also only shown to operate on lower-end setups (up to 20MHz SISO with 10× less throughput than what we show for Janus) and don't provide access to high throughput data streams such as IQ samples.

**Patching code with checkpoints -** Adding compiler-assisted checkpoints has been studied in the literature as a way of improving fault tolerance by periodically saving the software state (e.g., in intermittent energy systems) [67, 68, 80, 99, 101]. The choice of checkpoints in such systems is typically made with the goal of minimizing the energy overhead without affecting recoverability, which leads to different design choices compared to the pre-emption goal of Janus. Closer to Janus, the works in [95, 96] focus on adding checkpoints for asserting whether the allotted worst-case execution time of a real-time system has been exceeded. Contrary to Janus, such checks require hardware assistance and checkpoints are added in every basic block of the running program, which, as shown in §7.3 has a higher overhead compared to the Janus patcher.

**RAN data collection -** A number of works in the vRAN space offer solutions for data collection, ranging from API specifications (e.g., O-RAN RIC E2 service model [24, 85] and FlexRAN API [43]) to elaborate logging systems (e.g. OAI T-tracer [7], SCOPE data collection module [30])). However, such solutions offer no flexibility to adapt the type, volume and frequency of collected data based on the application's needs, which is one of the main design goals of Janus. Similar observations can be made for data collection solutions in the eBPF space, which either offer a fixed set of metrics (e.g. Hubble [5]) or data can only be exported in certain formats (e.g. counters and histograms as in ebpf_exporter [4]).

**Support for complex ML models using Janus maps-** As discussed in §4, the development of large ML-models (e.g., Random Forests) inline in Janus C code is challenging, mainly due to the big size of the generated bytecode and its implications to verification. We believe that our proposed map-based approach for loading ML models in a serialized format and performing inference through helper functions is powerful, considering that a large number of use cases proposed in the RAN space rely on the same models (e.g., Random Forests, LSTMs and RNNs [18, 40, 54, 66, 84, 89, 97, 98]). Therefore, we are planning on extending this approach to support additional widely-used ML models beyond Random Forests.

## 9 CONCLUSIONS

In this work we presented Janus, a fully programmable and safe monitoring and control framework for 5G RAN. It allows operators to load custom codelets with custom data models in real-time, significantly increasing flexibility offered by the existing O-RAN RIC. We demonstrated this flexibility by building and evaluating 17 applications in 4 different classes (most not achievable with O-RAN RIC). Janus achieves safety using static verification and codelet pre-emption. Its modular design makes it is easy to add to existing vRAN products. We hope that Janus will be eventually adopted by the O-RAN community to help accelerate innovation in the Open RAN.

## REFERENCES


[1] 2021. pyelftools. https://github.com/eliben/pyelftools.
[2] 2022. Awesome eBPF. https://github.com/zoidbergwill/awesome-ebpf.
[3] 2022. eBPF. https://ebpf.io/.
[4] 2022. ebpf_exporter. https://github.com/cloudflare/ebpf_exporter.
[5] 2022. Hubble exported metrics. https://docs.cilium.io/en/stable/operations/metrics/#hubble-exported-metrics.
[6] 2022. nanopb. https://github.com/nanopb/nanopb.
[7] 2022. OAI T Tracer. https://gitlab.eurecom.fr/oai/openairinterface5g/-/wikis/T.
[8] 2022. The LLVM compiler infrastructure. https://llvm.org/.
[9] O-RAN Working Group 3. 2021. Use Cases and Requirements. *O-RAN.WG3.UCR-v01.00* (2021).
[10] 3GPP. 2018. 3GPP TS 38.410: NG general aspects and principles. (2018).
[11] 3GPP. 2018. 3GPP TS 38.463: E1 Application protocol (E1AP). (2018).
[12] 3GPP. 2019. 3GPP TS 38.470: F1 general aspects and principles. (2019).
[13] 3GPP. 2020. 3GPP TS 28.552: Management and orchestration: 5G performance measurements. (2020).
[14] 3GPP. 2020. 3GPP TS 32.425: Performance Management (PM); Performance measurements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN). (2020).
[15] 3GPP. 2020. 3GPP TS 38.322: Radio Link Control (RLC) protocol specification. (2020).
[16] 3GPP. 2020. 3GPP TS 38.331: Radio Resource Control (RRC) protocol specification . (2020).
[17] 3GPP. 2020. 3GPP TS 38.420: Xn general aspects and principles. (2020).
[18] Javed Akhtar, Krunal Saija, Narayanan Ravi, Shekar Nethi, and Saptarshi Chaudhuri. 2021. Machine Learning-based Prediction of PMI Report for DL-Precoding in 5G-NR System. In *2021 IEEE 4th 5G World Forum (5GWF)*. IEEE, 105–110.
[19] ORAN Alliance. 2019. O-RAN WhitePaper-Building the Next Generation RAN. *O-RAN Alliance, Tech. Rep., Oct* (2019).
[20] ORAN Alliance. 2020. O-RAN use cases and deployment scenarios. *White Paper, Feb* (2020).
[21] ORAN Alliance. 2020. O-RAN Working Group 3: Near-Real-time RAN Intelligent Controller-E2 Service Model (E2SM). *ORAN-WG3. E2SM-KPM-v01. 00.00* (2020).
[22] ORAN Alliance. 2021. O-RAN Fronthaul Control User and Synchronization Plane Specification v7.0.
[23] ORAN Alliance. 2021. O-RAN Working Group 2: "O-RAN AI/ML workflow description and requirements 1.03. *O-RAN.WG2.AIML-v01.03 Technical Specification* (2021).
[24] O-RAN Alliance. 2021. O-RAN E2 Application Protocol (E2AP) v2.0. *ORAN-WG3.E2AP-KPM-v02.00* (2021).
[25] O-RAN Alliance. 2021. O-RAN Minimum Viable Plan and Acceleration towards Commercialization. *White Paper, June* (2021).
[26] OpenAir Software Alliance. 2022. Open Air Interface Project. https://openairinterface.org/about-us/.
[27] Bharath Balasubramanian, E Scott Daniels, Matti Hiltunen, Rittwik Jana, Kaustubh Joshi, Rajarajan Sivaraj, Tuyen X Tran, and Chengwei Wang. 2021. RIC: A RAN intelligent controller platform for AI-enabled cellular networks. *IEEE Internet Computing* 25, 2 (2021), 7–17.
[28] Andson Balieiro, Kelvin Dias, and Paulo Guarda. 2021. A Machine Learning Approach for CQI Feedback Delay in 5G and Beyond 5G Networks. In *2021 30th Wireless and Optical Communications Conference (WOCC)*. IEEE, 26–30.
[29] Gilberto Berardinelli, Saeed R Khosravirad, Klaus I Pedersen, Frank Frederiksen, and Preben Mogensen. 2016. Enabling early HARQ feedback in 5G networks. In *2016 IEEE 83rd Vehicular Technology Conference (VTC Spring)*. IEEE, 1–5.
[30] Leonardo Bonati, Salvatore D'Oro, Stefano Basagni, and Tommaso Melodia. 2021. SCOPE: an open and softwarized prototyping platform for NextG systems. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 415–426.
[31] Leonardo Bonati, Salvatore D'Oro, Michele Polese, Stefano Basagni, and Tommaso Melodia. 2021. Intelligence and learning in O-RAN for data-driven NextG cellular networks. *IEEE Communications Magazine* 59, 10 (2021), 21–27.
[32] Gabriel Brown and HEAVY READING. 2018. New Transport Network Architectures for 5G RAN. *White Paper. Available online: https://www. fujitsu. com/us/Images/New-Transport-Network-Architectures-for-5G-RAN. pdf (accessed on 29 June 2021)* (2018).
[33] Nicola Bui and Joerg Widmer. 2018. Data-driven evaluation of anticipatory networking in LTE networks. *IEEE Transactions on Mobile Computing* 17, 10 (2018), 2252–2265.
[34] Cambridge Consultants. 2022. Wireless breakthrough for the Ocado Smart Platform. https://www.cambridgeconsultants.com/case-studies/wireless-breakthrough-ocado-smart-platform.
[35] CapGemini Engineering. 2022. 5G gNodeB. https://capgemini-engineering.com/nl/en/services/next-core/wireless-frameworks/.
[36] Francisco J Cazorla, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella, and Tullio Vardanega. 2019. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Computing Surveys (CSUR)* 52, 1 (2019), 1–35.
[37] Cellwize and Intel. 2022. Cellwize Announces Collaboration to Accelerate Deployment of 5G vRAN Networks With AI. https://www.sdxcentral.com/articles/press-release/cellwize-announces-collaboration-to-accelerate-deployment-of-5g-vran-networks-with-ai/2021/06/.
[38] Cohere Technologies. 2022. With Vodafone and Partners, VMWare demonstrates how to accelerate innovation in the RAN. https://www.cohere-tech.com/press-releases/with-vodafone-and-partners-vmware-demonstrates-how-to-accelerate-innovation-in-the-ran.
[39] Salah Eddine Elayoubi, Sana Ben Jemaa, Zwi Altman, and Ana Galindo-Serrano. 2019. 5G RAN slicing for verticals: Enablers and challenges. *IEEE Communications Magazine* 57, 1 (2019), 28–34.
[40] Capgemini Engineering. 2021. Intelligent 5G L2 MAC Scheduler. *White Paper, Feb* (2021).
[41] Small Cell Forum. 2021. 5G FAPI: PHY API Specification.

[42] Xenofon Foukas, Mahesh K Marina, and Kimon Kontovasilis. 2019. Iris: Deep reinforcement learning driven shared spectrum access architecture for indoor neutral-host small cells. *IEEE Journal on Selected Areas in Communications* 37, 8 (2019), 1820–1837.

[43] Xenofon Foukas, Navid Nikaein, Mohamed M Kassem, Mahesh K Marina, and Kimon Kontovasilis. 2016. FlexRAN: A flexible and programmable platform for software-defined radio access networks. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 427–441.

[44] Xenofon Foukas, Georgios Patounas, Ahmed Elmokashfi, and Mahesh K Marina. 2017. Network slicing in 5G: Survey and challenges. *IEEE communications magazine* 55, 5 (2017), 94–100.

[45] Xenofon Foukas and Bozidar Radunovic. 2021. Concordia: teaching the 5G vRAN to share compute. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 580–596.

[46] Gines Garcia-Aviles, Andres Garcia-Saavedra, Marco Gramaglia, Xavier Costa-Perez, Pablo Serrano, and Albert Banchs. 2021. Nuberu: Reliable RAN virtualization in shared platforms. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*. 749–761.

[47] Andres Garcia-Saavedra and Xavier Costa-Perez. 2021. O-RAN: Disrupting the virtualized RAN ecosystem. *IEEE Communications Standards Magazine* (2021).

[48] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1069–1084.

[49] David Ginthör, René Guillaume, Maximilian Schüngel, and Hans D Schotten. 2021. 5G RAN slicing for deterministic traffic. In *2021 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 1–6.

[50] Google. 2022. Sandbox2. https://developers.google.com/code-sandboxing/sandbox2.

[51] Google. 2022. Sandboxed API. https://developers.google.com/code-sandboxing/sandboxed-api.

[52] Tao Guo and Alberto Suárez. 2019. Enabling 5G RAN slicing with EDF slice scheduling. *IEEE Transactions on Vehicular Technology* 68, 3 (2019), 2865–2877.

[53] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.

[54] Sahar Imtiaz, Georgios P Koudouridis, Hadi Ghauch, and James Gross. 2018. Random forests for resource allocation in 5G cloud radio access networks based on position information. *EURASIP Journal on Wireless Communications and Networking* 2018, 1 (2018), 1–16.

[55] Intel. 2020. Unleash the Speed of 4G and 5G Virtualized Radio Access Networks (vRAN). *Product Brief, Intel vRAN Dedicated Accelerator ACC100* (2020).

[56] Intel. 2020. Virtual RAN (vRAN) with Hardware Acceleration. *White Paper, Jan* (2020).

[57] Intel. 2022. FlexRAN Reference Architecture for Wireless Access. https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/tools/flexran.html.

[58] Intel. 2022. Smart Edge Open Radio Access Network. https://smart-edge-open.github.io/ido-specs/doc/reference-architectures/ran/smartedge-open_ran/.

[59] iovisor. 2022. Userspace eBPF VM. https://github.com/iovisor/ubpf.

[60] David Johnson, Dustin Maas, and Jacobus Van Der Merwe. 2022. NexRAN: Closed-loop RAN slicing in POWDER-A top-to-bottom open-source open-RAN use case. In *Proceedings of the 15th ACM Workshop on Wireless Network Testbeds, Experimental evaluation & CHaracterization*. 17–23.

[61] Ravi Kokku, Rajesh Mahindra, Honghai Zhang, and Sampath Rangarajan. 2011. NVS: A substrate for virtualizing wireless resources in cellular networks. *IEEE/ACM transactions on networking* 20, 5 (2011), 1333–1346.

[62] Manikanta Kotaru, Kiran Joshi, Dinesh Bharadia, and Sachin Katti. 2015. Spotfi: Decimeter level localization using wifi. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 269–282.

[63] Merima Kulin, Tarik Kazaz, Ingrid Moerman, and Eli De Poorter. 2018. End-to-end learning from spectrum data: A deep learning approach for wireless signal identification in spectrum monitoring applications. *IEEE Access* 6 (2018), 18484–18501.

[64] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of {WebAssembly}. In *29th USENIX Security Symposium (USENIX Security 20)*. 217–234.

[65] liburcu. 2022. Userspace RCU. https://liburcu.org/.

[66] Changqing Luo, Jinlong Ji, Qianlong Wang, Xuhui Chen, and Pan Li. 2018. Channel state information prediction for 5G wireless communications: A deep learning approach. *IEEE Transactions on Network Science and Engineering* 7, 1 (2018), 227–236.

[67] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.

[68] Kiwan Maeng and Brandon Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 129–144.

[69] Paul E McKenney, Silas Boyd-Wickizer, and Jonathan Walpole. 2013. RCU usage in the linux kernel: One decade later. *Technical report* (2013).

[70] Paul E McKenney and Jonathan Walpole. 2007. What is RCU, fundamentally? *Linux Weekly News (LWN. net)* (2007).

[71] Mehrtash Mehrabi, Mostafa Mohammadkarimi, Masoud Ardakani, and Yindi Jing. 2019. Decision Directed Channel Estimation Based on Deep Neural Network $k$-Step Predictor for MIMO Communications in 5G. *IEEE Journal on Selected Areas in Communications* 37, 11 (2019), 2443–2456.

[72] Mustafa Mohsin, Jordi Mongay Batalla, Evangelos Pallis, George Mastorakis, Evangelos K Markakis, and Constandinos X Mavromoustakis. 2021. On Analyzing Beamforming Implementation in O-RAN 5G. *Electronics* 10, 17 (2021), 2162.

[73] Navid Nikaein, Chia-Yu Chang, and Konstantinos Alexandris. 2018. Mosaic5G: Agile and flexible service platforms for 5G research. *ACM SIGCOMM Computer Communication Review* 48, 3 (2018), 29–34.

[74] Daisuke Nojima, Yuki Katsumata, Takuya Shimojo, Yoshifumi Morihiro, Takahiro Asai, Akira Yamada, and Shigeru Iwashina. 2018. Resource isolation in RAN part while utilizing ordinary scheduling algorithm for network slicing. In *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*. IEEE, 1–5.

[75] O-RAN SC projects . 2022. RAN Intelligent Controller Applications. https://docs.o-ran-sc.org/en/latest/projects.html#ran-intelligent-controller-applications-ricapp.

[76] Gabriele Paoloni. 2010. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. *Intel Corporation* 123 (2010), 170.

[77] Michele Polese, Leonardo Bonati, Salvatore D'Oro, Stefano Basagni, and Tommaso Melodia. 2022. Understanding O-RAN: Architecture, Interfaces, Algorithms, Security, and Research Challenges. *arXiv preprint arXiv:2202.01032* (2022).

[78] Michele Polese, Rittwik Jana, Velin Kounev, Ke Zhang, Supratim Deb, and Michele Zorzi. 2020. Machine learning at the edge: A data-driven architecture with applications to 5G cellular networks. *IEEE Transactions on Mobile Computing* 20, 12 (2020), 3367–3382.
[79] Protocol Buffers. 2022. Protobuf benchmark proto file. https://github.com/protocolbuffers/protobuf/blob/fb77cc9d9f066a8ce4f12e8d5f76188d48101444/benchmarks/google_size.proto.
[80] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System support for long-running computation on RFID-scale devices. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. 159–170.
[81] Raymond Knopp. 2022. OAI Layer 2 Protocol Stack. https://www.openairinterface.org/docs/workshop/1stOAINorthAmericaWorkshop/Training/KNOPP-OAI-L2.pdf.
[82] Rohde & Schwarz. 2022. PREEMPT_RT patch versions. https://www.rohde-schwarz.com/uk/solutions/aerospace-defense-security/security/spectrum-monitoring/efficient-interference-hunting/huntinginterferences_91389.html.
[83] Peter Rost and Athul Prasad. 2014. Opportunistic hybrid ARQ—Enabler of centralized-RAN over nonideal backhaul. *IEEE Wireless Communications Letters* 3, 5 (2014), 481–484.
[84] Krunal Saija, Shekar Nethi, Saptarshi Chaudhuri, and RM Karthik. 2019. A machine learning approach for SNR prediction in 5G systems. In *2019 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*. IEEE, 1–6.
[85] Robert Schmidt, Mikel Irazabal, and Navid Nikaein. 2021. FlexRIC: an SDK for next-generation SD-RANs. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. 411–425.
[86] Souvik Sen, Božidar Radunovic, Romit Roy Choudhury, and Tom Minka. 2012. You are facing the Mona Lisa: Spot localization using PHY layer information. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 183–196.
[87] Weisen Shi, Junling Li, Peng Yang, Qiang Ye, Weihua Zhuang, Xuemin Shen, and Xu Li. 2021. Two-level soft RAN slicing for customized services in 5G-and-beyond wireless communications. *IEEE Transactions on Industrial Informatics* 18, 6 (2021), 4169–4179.
[88] Small Cell Forum. 2022. 5G FAPI: PHY API Specification. http://scf.io/en/documents/222_5G_FAPI_PHY_API_Specification.php.
[89] Nils Strodthoff, Barış Göktepe, Thomas Schierl, Cornelius Hellge, and Wojciech Samek. 2019. Enhanced machine learning techniques for early HARQ feedback prediction in 5G. *IEEE Journal on Selected Areas in Communications* 37, 11 (2019), 2573–2587.
[90] Telefonica. 2021. Telefonica views on the design, architecture, and technology of 4G/5G Open RAN networks. *White Paper, Jan* (2021).
[91] TensorFlow. 2022. TensorFlow Lite for Microcontrollers. https://www.tensorflow.org/lite/microcontrollers.
[92] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. 2020. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.
[93] OpenAirInterface Wiki. 2022. OpenAirKernelMainSetup. https://gitlab.eurecom.fr/oai/openairinterface5g/-/wikis/OpenAirKernelMainSetup.
[94] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 1–53.
[95] Julian Wolf, Bernhard Fechner, Sascha Uhrig, and Theo Ungerer. 2012. Fine-grained timing and control flow error checking for hard real-time task execution. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. IEEE, 257–266.
[96] Julian Wolf, Bernhard Fechner, and Theo Ungerer. 2012. Fault coverage of a timing and control flow checker for hard real-time systems. In *2012 IEEE 18th International On-Line Testing Symposium (IOLTS)*. IEEE, 127–129.
[97] Hao Yin, Xiaojun Guo, Pengyu Liu, Xiaojun Hei, and Yayu Gao. 2020. Predicting Channel Quality Indicators for 5G Downlink Scheduling in a Deep Learning Approach. *arXiv preprint arXiv:2008.01000* (2020).
[98] Qi Zhang, Alexandros Nikou, and Marios Daoutis. 2022. Predicting Buffer Status Report (BSR) for 6G Scheduling using Machine Learning Models. In *2022 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 632–637.
[99] Ying Zhang and Krishnendu Chakrabarty. 2003. Energy-aware adaptive checkpointing in embedded real-time systems. In *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 918–923.
[100] Yadan Zheng, Shubo Ren, Xiaoyan Xu, Ying Si, Mingke Dong, and Jianjun Wu. 2012. A modified ARIMA model for CQI prediction in LTE-based mobile satellite communications. In *2012 IEEE International Conference on Information Science and Technology*. IEEE, 822–826.
[101] Avi Ziv and Jehoshua Bruck. 1997. An on-line algorithm for checkpoint placement. *IEEE Transactions on computers* 46, 9 (1997), 976–985.

# A APPENDIX

## A.1 Loading Janus codeletsets and codelet coordination

As already explained in Section 3.1, Janus codeletsets are JIT-compiled and loaded to Janus devices, through a network API provided by the Janus controller. Developers can upload the developed codeletsets to the Janus controller and load them to Janus devices using a tool provided by the Janus SDK, as illustrated in Fig. 10. The instructions required to upload a codeletset to the Janus controller and load them to a Janus device are encoded in a descriptor file written in YAML format. The YAML structure and properties of codeletsets are listed in Listing 6.

```yaml
codelet1:
  codelet: codelet1.o
  hook_name: hook_name1
  priority: codelet1_running_priority
  runtime_threshold: codelet1_max_runtime
codeletN:
  codelet: codeletN.o
  hook_name: hook_nameN
  priority: codeletN_running_priority
  runtime_threshold: codeletN_max_runtime
  linked_maps:
    codeletN_map:
      codelet: codelet1
      map_name: codelet1_map
```

**Listing 6: Codeletset YAML structure**

At the minimum, the YAML file specifies the names of all the codelets that are part of the codeletset (eBPF bytecode

object files in ELF format), the name of the hook in the Janus device(s) they should be linked to and their calling priority (multiple codelets can be linked to the same hook). The YAML file provides additional optional configuration parameters for further configuring codelets, like for example the runtime threshold of each patched codelet when using the runtime control mechanism of Section 5.1.

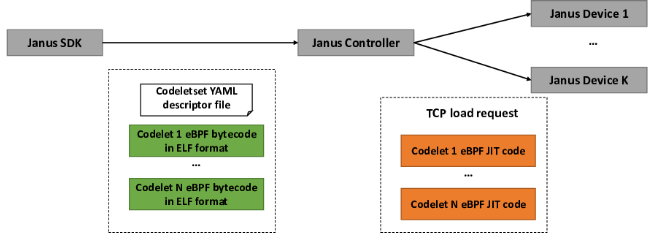



**Figure 10: Loading process of Janus codeletsets to Janus devices**

Janus codelets of the same codeletset can share state and coordinate. This can be particularly useful when performing monitoring or control operations that require the correlation of events and data across different layers of the vRAN stack (e.g. as in the case of interference detection in §4). The sharing of state between codelets of a codeletset is performed using shared maps. For this mechanism to work, codelets that want to share state must use the exact same definition for the shared map in terms of its `type`, `key_size`, `value_size` and `max_entries`, while the name of the map can be different. Developers can then specify the maps to be linked, in the `linked_maps` section of the YAML descriptor file they use for loading the codeletset, as shown in Listing 6. When the codeletset is loaded, memory for the shared map is only allocated once on the Janus device and all codelets sharing the map get a pointer to the same memory, which they can then call through helper functions to store and load state.

## A.2 Flexible output Janus schemas

Output schemas can be reused by multiple codelets. Therefore, the schemas and the codelets are uploaded to the Janus controller separately, using the Janus SDK. Codelets specify which output schema(s) they make use of in the definition of their output ringbuffer map(s). This includes three fields, as shown in lines 11-13 of Listing 1. The `proto_name` field contains a name that indicates a *unique* protobuf specification file that has been uploaded to the Janus controller. The `proto_msg_name` field indicates the root message of the `proto_name` specification that will be used to send data using this ringbuffer (a proto spec can contain multiple message definitions). Finally, the `proto_hash` field contains a hash value that is used to ensure that the contents of the protobuf specification file used during the development of the codelet are the same as those of the file uploaded to the Janus controller.

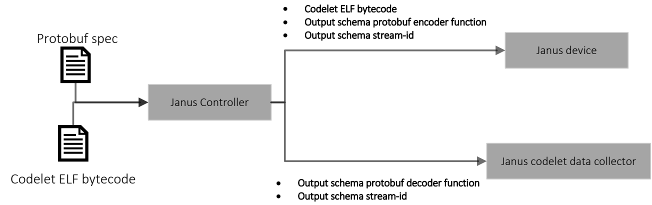



**Figure 11: Loading process of Janus codeletsets with output schemas to Janus devices**

Fig 11 illustrates the loading process of codeletsets with output schemas. The Janus controller parses the codelet object (ELF) files to identify maps of type `JANUS_MAP_TYPE_RINGBUF` in the *maps* section. It then uses the `proto_name` value in the map definition to link the codelet with some .proto specification file that has been previously uploaded to the controller. Once the specification is found, the controller compares the `proto_hash` field of the map with a hash digest of the .proto file. If those match, the controller assigns a unique *stream-id* to the codelet's ringbuf map (a 16-bytes UUID). Next, the controller auto-generates an *encoder* and a *decoder* function, that are responsible for the serialization and desirialization of messages of type `proto_msg_name` that are sent by the codelet to the Janus output data collector. The controller sends the auto-generated *encoder* function to the Janus device via the network API, along with the verified codelet's bytecode and the output map's *stream-id*. The controller also sends the *stream-id* along with the auto-generated *decoder* function to the output data collector. The collector maintains a key-value structure, that maps the *stream-id* (key) to the *decoder* function (value).

Each time a codelet with a `JANUS_MAP_TYPE_RINGBUF` output map is loaded to a Janus device, a single-producer/single-consumer (SPSC) ringbuffer data structure is created and linked to it, where the codelet is the producer and an *output thread* is the consumer. As illustrated in Fig. 12, every time that the codelet calls the `janus_ringbuf_output()` helper function (e.g., line 42 of Listing 1), the output data is pushed to the ringbuffer. The output thread consumes the data and calls the corresponding *encoder* function to serialize them. Before sending the serialized data to the output collector via UDP, a header is appended, that includes the 16-bytes *stream-id* and a 2-bytes sequence number. Once the output collector receives the output message, it matches the *stream-id* to the appropriate *decoder* function, which it uses to deserialize the message. Finally, the deserialized message is converted to JSON format and can then be fed to other components of the pipeline (e.g. storage, ML processing etc).

## A.3 Writing Janus hooks

To simplify the process of writing Janus hooks, the Janus SDK provides a set of macros for declaring and running new hooks. In short, the macro `DECLARE_JANUS_HOOK()` is called

| Function name | Description |
|---|---|
| hook_fapi_dl_config_req() | Runs all the codelets registered on the hook one by one based on their execution priorities |
| register_janus_codelet_fapi_dl_config_req() | Registers a codelet to the hook with a default execution priority |
| register_janus_codelet_prio_fapi_dl_config_req() | Registers a codelet to the hook with a user-defined execution priority |
| remove_janus_codelet_fapi_dl_config_req() | Removes a registered codelet from the hook |

**Table 5: Functions generated by `DECLARE_JANUS_HOOK()` macro for example of Listing 7.**

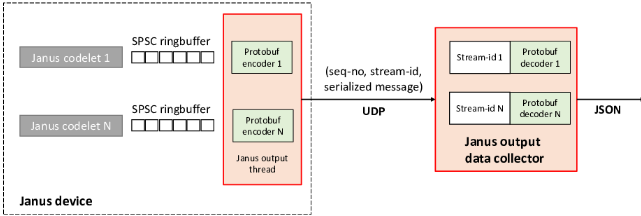



**Figure 12: Exporting of data from Janus codelets**

and takes as input arguments the name of the new hook, the context type that will be passed to the codelets called by the hook, the signature of the hook function (name and type of arguments), as well as a list of assignments for populating the context that will be passed to the codelet. An example for the codelet of Listing 1 is shown in Listing 7.

```
1  DECLARE_JANUS_HOOK(fapi_dl_config_req,
2         struct janus_ran_fapi_ctx ctx,
3         ctx,
4         HOOK_PROTO(
5              nfapi_dl_config_request_pdu_t *dl_config_req,
6              int ctx_id,
7              int frame,
8              int subframe,
9              int cell_id,
10             int fapi_list_size
11             ),
12        HOOK_ASSIGN(
13             ctx.ctx_id = ctx_id;
14             ctx.cell_id = cell_id;
15             ctx.slot = subframe;
16             ctx.frame = frame;
17             ctx.data = (void *) dl_config_req;
18             ctx.data_end = (void *) (dl_config_req +
       fapi_list_size);
19             )
20        )
```

**Listing 7: Hook declaration for codelet of Listing 1**

Based on those inputs, the macro auto-generates boilerplate code for the functions that form the API of the hook. This includes functions for loading and unloading codelets to the hook, as well as a function to run all the codelets that are linked with the hook. A list of the autogenerated functions for the example of Listing 7 is shown in Table 5.

Once a hook is declared, developers can simply introduce the hook in their vCU/vDU code by instantiating it using a macro called `DEFINE_JANUS_HOOK()`, passing as an argument the name of the hook. Finally, the hook can be called at any point of the code by calling the auto-generated function `hook_#hook_name()`, where `#hook_name` is the name that was used when declaring the hook. For example, in the case of Listing 7, a hook called (e.g. `hook_fapi_dl_config_req()`) is generated.

## A.4 Inference using Janus maps

Here, we explain in more details how Janus can use more complex ML models for inference using maps. A simple example codelet is shown in Listing 8 for the Random Forest model that was discussed in §4. This codelet performs inference for a pre-trained model and then simply returns.

```
1  struct janus_load_map_def SEC("maps") model_map = {
2    .type = JANUS_MAP_TYPE_ML_MODEL,
3    .max_entries = 16,
4    .ml_model = "random_forest",
5  };
6
7  struct features {
8    int f1;
9    int f2;
10   int f3;
11   int o1;
12 };
13
14 SEC("janus_ran_fapi")
15 uint64_t bpf_prog(void *state) {
16
17   struct features feats;
18   int res;
19
20   feats.f1 = 1;
21   feats.f2 = 1;
22   feats.f3 = 1;
23
24   /* We store inference result in output */
25   res = janus_model_predict(&model_map, &feats);
26
27   if (res) return 1;
28
29   return 0;
30 }
```

**Listing 8: ML model usage example**

**Loading an ML model -** As it can be seen in line 2 of Listing 8, we define a map of type `JANUS_MAP_TYPE_ML_MODEL`. The model must be loaded by the controller to janus in a serialized format from an input file called `random_forest` (line 4). The serialized model is represented as a char array and captures all the important information required by janus to recreate the trained model in memory. For example, for the random forest model, the serialization contains information about the type of model (random forest), the number of estimators (trees), the condition to check on each node of each estimator, as well as the inference values of the leaf nodes of the trees. Janus only supports a pre-determined set of model types (currently random forest and SVM), each with its own set of serialization parameters. During the loading of the codelet, the verifier will check if the loaded serialized model is valid. This includes checks for the type of model as well as the validity of the model parameters (i.e., whether the model can be

reconstructed in memory). If any of the checks fails, then the codelet is not loaded in the Janus device.

**In-codelet inference -** The loaded model expects a set of input features and outputs of known number and size. The required memory for the input features and the outputs is specified in the definition of the ML model map. For the example of Listing 8, we have 3 input features and one output (lines 7-12) for a total memory size of 16 Bytes (line 3). The exact memory layout of the input features and the outputs depends on the trained model and is thus codelet specific. Using the input features, we can perform inference as shown in line 25, by calling the helper function `janus_model_predict()`.

## A.5 Real-time operation

Here, we provide more details about optimizations that we made for Janus in order to ensure real-time performance:

**Output maps -** For each output stream, a lock-free single-producer/single-consumer ring buffer is created to push data out from the codelet to the output thread without making any system calls (see Fig. 2). This ensures that the codelet will never be pre-empted and might only drop excess packets in the worst case.

**Memory allocation -** Janus uses pre-allocated memory for its operation (JIT code of loaded codelets, maps, data output etc.). It relies on DPDK mempools and Mbufs operating in lock-free mode (using the `rte_stack` mempool handler). This ensures that multiple threads using the same mempool will not affect the performance of the rest if pre-empted, minimizing the jitter when acceessing memory in the fast-path.

**Concurrency of Janus maps -** By design, Janus maps do not use locks, in order to guarantee the real-time performance of the time critical vRAN functions, and are thus not thread-safe. This could cause concurrency issues (e.g. for hooks called in the code of worker threads with multiple instances). However, based on our experience using Janus with commercial-grade vRAN functions, we believe that in most cases codelets can be written in a lockless way, as long as some kind of context id is passed as part of the hook context, to identify which instance is calling the hook (e.g. which CPU core, which worker thread etc.). We are planning on relaxing this constraint in the future through thread-safe maps, where appropriate.

**Loading/unloading codelets -** We used a userspace implementation of the Read-Copy-Update mechanism (RCU) [65] for (un)loading codelets into Janus hooks. RCU allows multiple uncoordinated readers to access a shared data structure at the expense of a longer write/update time [69, 70]. This fits well with the Janus design, where the readers are the fast vRAN threads that execute codelets, and the writer is the (non real-time) thread that updates the hook codelet list.