Taking 5G RAN Analytics and Control to a New Level

Xenofon Foukas, Bozidar Radunovic, Matthew Balkwill, Zhihua Lai
{xefouk,bozidar,mabalkwi,zhihualai}@microsoft.com
Microsoft

ABSTRACT

Open RAN, a modular and disaggregated design paradigm for 5G radio access networks (RAN), promises programmability through the 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 inline 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 (18 applications in total) and deploying them on a 100MHz 4x4 MIMO 5G cell without affecting the RAN performance.

CCS CONCEPTS

• **Networks** → **Mobile networks**; **Wireless access points, base stations and infrastructure**; **Programmable networks**.

KEYWORDS

vRAN, 5G, mobile networks, programmability, RIC

ACM Reference Format:
Xenofon Foukas, Bozidar Radunovic, Matthew Balkwill, Zhihua Lai. 2023. Taking 5G RAN Analytics and Control to a New Level. In *The 29th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '23), October 2–6, 2023, Madrid, Spain.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3570361.3592493

ACM MobiCom '23, October 2–6, 2023, Madrid, Spain

ACM ISBN 978-1-4503-9990-6/23/10…$15.00
https://doi.org/10.1145/3570361.3592493

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) [38, 42], allows 3rd parties to optimize the network by building data-driven, vendor-agnostic monitoring and control applications [19, 30] over open interfaces standardized by O-RAN [29].

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 [23, 24], defines a small set of aggregate Key Perfomance Indicators (KPIs) collected every few seconds or minutes. The O-RAN RIC extends this idea with a new set of aggregate KPIs and data sources [36]. Each KPI is defined through a service model (a static API that is embedded in the vRAN functions [35]) 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.

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 μs 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 [89]. The existing RIC approach deals with this issue by specifying service models tailored to specific use cases, each with a supported set of policies (choose one out of N available algorithms). However, this also 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 [56] 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 the limitations that arise from the static nature of the existing RIC service models, 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 executed inline, allowing them to get direct access to raw vRAN data structures, to collect arbitrary statistics and to make real-time inference and control decisions.

While Janus significantly enhances the RIC capabilities, it also comes with its own challenges. The first has to do with flexibility. It is unclear which RAN monitoring data and control knobs should be exposed to developers to build useful apps. We solve this by identifying key locations and interfaces (we call them *hooks*) within the vRAN architecture that provide rich data and unlock a wide range of control applications. We also build a toolchain that allows developers to define arbitrary output schemas to ship the collected data to the RIC. We show in §4 that Janus codelets can be used to implement O-RAN service models 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 [5, 105], which solves a similar problem in the Linux kernel [3]. Codelets are written in C and are compiled into eBPF bytecode. The bytecode runs inside a virtual environment inlined in the vRAN's control and data path, with direct access to selected internal RAN data structures and control functions. Prior to loading a codelet, the eBPF bytecode is statically verified [60, 105] and only codelets that are safe in terms of memory accesses are allowed to run.

We further extend this model to tailor it to the vRAN requirements. We introduce hard, μs-level control in the execution latency of codelets through an eBPF bytecode patching mechanism that preempts a codelet that exceeds a certain runtime threshold. Furthermore, we extend the static verification to cover the newly introduced flexible output data structures and we provide several optimizations to ensure a non-preemtible design in the fast path, minimizing Janus' impact to the performance of the vRAN. Finally, we integrate Janus with a commercial 5G vRAN stack from CapGemini [46] (based on the Intel FlexRAN reference design [69]) and with the open source 4G/5G stack of OpenAirInterface (OAI) [37].

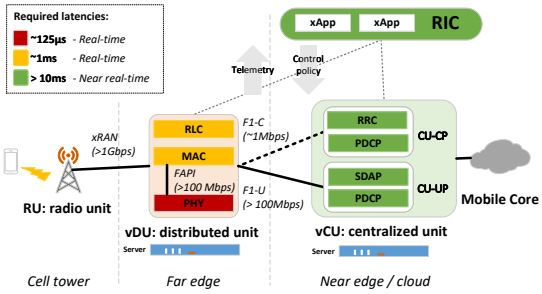


Figure 1: High-level vRAN architecture and processing and throughput requirements of vRAN functions.

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 (§3). We illustrate its functionality by developing new telemetry, control and inference applications (18 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).
- We present a concrete and optimized implementation of Janus (§6) and perform a thorough evaluation (§7).

We hope that Janus will gather O-RAN community support and will be integrated with the O-RAN RIC in the future.

2 BACKGROUND & MOTIVATION

2.1 vRAN Architecture

The 5G RAN consists of several *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 control plane and user-plane (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 [68, 103]. Different components have different latency requirements(c.f. [43]) 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 3GPP, O-RAN [29] and the Small Cell Forum [52], and programmability is facilitated through a near real-time RIC [59]. Network operators install applications (xApps in the

O-RAN terminology) on the RIC to collect data and leverage it for monitoring, inference, and near real-time ($> 10ms$) closed loop control. Data collection and control is facilitated through *service models* that are embedded in the vRAN functions by vendors and define the xApps' capabilities in terms of the type and frequency of data reporting and supported control policies.

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 [72, 84, 89, 97]. 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 [74], channel estimation [77, 80], interference detection [75] and beamforming [81] 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 frequency and type (e.g., sub-sampling vs. averages) in the service model of each xApp (e.g., as in [48, 49]). This poses a serious limitation to interoperability, since vRAN vendors must implement and support each proprietary service model.

Real-time limitations: Some vRAN control loops, like UE radio resource allocation, have tight time constraints (10s of μs to a few ms). Such time constraints cannot be met by the current RIC design, that has an expected latency > 10ms [89]. xApps overcome this issue by using a set of pre-defined policies offered by service models, which can run inline inside the vRAN functions. However, this approach doesn't scale as the number of policies increases. For example, several control algorithms have been proposed for network slicing (e.g., [53, 61, 64, 73, 83, 99]), 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

We argue that, to unlock the true RIC capabilities, a new solution is needed, which 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 deadline violations.

[1] It requires more than 1.5 Gbps per cell for 100 MHz 4 × 4 MIMO

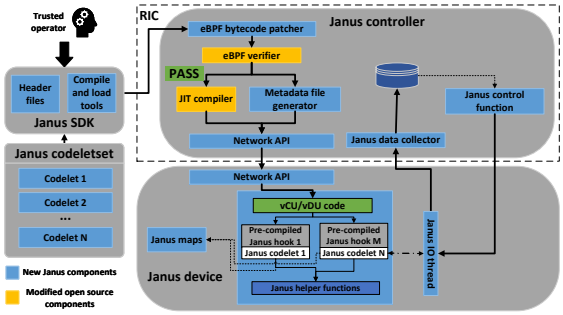


Figure 2: The high-level architecture of Janus.

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) that allows execution of custom code. Janus introduces 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, which includes various 3GPP-defined data structures and events (see Table 1 and §3.2). The type of data that is passed to a codelet depends on the layer the hook is introduced to and could include packets of users, signaling messages etc. 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), provides static code verification and has been met with widespread success in several networking projects (e.g., [4, 8, 13, 15]). For more technical details about eBPF, we refer the interested reader to [17, 18]. Other approaches that we considered, such as Sandbox2 [62] and SAPI [63], run custom code in separate processes incurring extra IPC latency. WebAssembly [65] is inlined, but its lack of static verification can lead to memory violation issues [76].

Janus codelets: A Janus *codelet* is a custom code that can be deployed at a single hook at runtime. Developers write codelets in C and compile them into eBPF bytecode. Similar to any eBPF program, a Janus codelet must be statically verifiable

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 [32]
FAPI interface	vDU	Capture scheduling and data plane packets exchanged between MAC and PHY layers [52]
RLC	vDU	Capture information about buffers of mobile devices and RLC mode/parameters [25]
F1/E1/Ng/Xn interfaces	vCU/vDU	Capture control/data-plane messages exchanged between 3GPP interfaces of vCU/vDU/5G core [20–22, 27]
RRC	vCU	Capture RRC messages exchanged between mobile devices and the base station [26]

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

(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 allocated to the codelet) can only be performed 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. Similarly, a codelet can receive control commands from a control application running at the Janus controller. The command is received by a controller thread and is then pushed to a special map, to be received by the codelet. 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. 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 the safety and termination of each codelet. The controller further instruments the verified bytecode with control code that pre-empts it if its runtime exceeds some threshold (see Section 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 and input of control commands using protobuf schemas (see Section 5.2). The controller provides a data collector, which collects and deserializes the data sent from the Janus codelets. It also provides an API that allows control applications to send arbitrary control commands to loaded Janus codelets (see Section 5.2). Janus also provides an SDK that 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 codelet developed for the vDU of OpenAirInterface [37]. The codelet is invoked by a hook that is introduced at the FAPI interface ([101], 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 a Janus hook and contains a pointer to a FAPI structure [52, 92] (line 24). It describes the scheduling allocation for a particular downlink slot, comprised of more than 20 fields per user, including transport block size, allocated resource blocks, MIMO etc. The verifier ensures read-only access to the context. Due to the modular vRAN design, there is a small number of similar standardized interfaces specified (3GPP, Small cell forum, O-RAN) that carry all relevant state across vRAN components. By adding hook points at these interfaces we can give developers access

Hook point	Type of control
xRAN packet transmission/reception	xRAN fronthaul procedure [32] – Drop/Forward/Modify xRAN packets sent and received by the radio unit
MAC scheduler invocation	MAC scheduling procedure [52] – Set modulation & coding scheme, uplink power and allocated resource blocks of UEs
RRC event handler (Triggered by event or timer)	Radio Resource Management-related procedures [26, 34] – Modify/Forward/Drop/Generate RRC message

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

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 several applications without modifying a single line of code inside the vRAN functions.

Statefulness: Janus codelets rely on shared memory regions known as *maps* to store state across consecutive invocations and to exchange state with other codelets. 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).

Safe & expressive custom control operations: Janus allows the introduction of custom control logic in the vRAN network functions. As with telemetry, we propose the introduction of control hooks in a small number of well-defined locations introduced by the relevant standardization bodies: i) fronthaul traffic control [32], ii) MAC scheduling control [52], and iii) Radio Resource Management [26, 34]. Janus control hooks follow an approach similar to XDP for Linux [105]. Codelets are allowed to modify the input state passed by the control hook as part of the context (e.g., modify a fronthaul packet header or generate/modify a MAC scheduling decision). Only a single codelet is allowed to be loaded at each control hook. Along with the context/packet modifications, codelets must provide a return code, which is used by the vRAN vendor to decide what action to take (e.g., forward/drop packet, ignore/apply scheduling decision etc.). As a first step towards this direction, we implemented a small number of control hooks, listed in Table 2. The hooks provide a subset of the envisioned control capabilities, but can already enable a large range of novel applications, which we present in more details in §4. As an example, we used the *MAC scheduler invocation* hook point to implement 3 real-time network slicing algorithms from the literature in OAI.

Flexible 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 Section 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. We apply the same mechanism to enable external control applications (e.g., applications running on the RIC) to dynamically modify the behavior of codelets (e.g. the values of some codelet parameters) by sending control messages using custom user-defined APIs (see Section 5.2).

4 NOVEL JANUS USE CASES

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

Flexible monitoring: We use Janus to implement codelets that extract KPIs specified in the KPM model of O-RAN [31] (lines 1-8 in Table 3), as well as raw scheduling data (lines 9-10) *without changing 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 [42, 59] on the fly, without undergoing a lengthy standardization process. For example, we were able to collect the downlink total Physical Resource Block (PRB) usage KPI [23], by tapping into the FAPI hook of Table 1 and capturing the number of PRBs allocated to each user at each scheduling decision using the `nfapi_dl_config_request_pdu_t` struct (Listing 1). The data of this struct were stored in a Janus map, averaged over a 0.5ms period and sent 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 an application that allows us to detect external radio interference by transforming an operational 5G radio unit to a spectrum sensor. Our application leverages a codeletset for the data collection, which is composed of two codelets that use maps for coordination. The first detects idle resource blocks when there are no 5G transmissions (installed at the FAPI hook of Table 1) and the second extracts IQ samples from the observed idle resource blocks (installed at the IQ samples hook of Table 1) and exports them to the application. The application detects and reports interference, if the energy level of the (unused) IQ samples exceeds a certain threshold. The flexibility of Janus allows us to adjust the fidelity and

Line	Name	Type	Benefit	LOC	Short	Ref
1	Total DL PRB Usage	M	Data aggregation without introducing new service model	207	KPM1	5.1.1.2.1 [23]
2	Total UL PRB Usage	M		171	KPM2	5.1.1.2.2 [23]
3	Distr. of DL PRB Usage	M		232	KPM3	5.1.1.2.3 [23]
4	Distr. of UL PRB Usage	M		197	KPM4	5.1.1.2.4 [23]
5	Total num. of initial DL TBs	M		152	KPM5	5.1.1.7.1 [23]
6	Total num. of DL TBs	M		156	KPM6	5.1.1.7.3 [23]
7	Total num. of initial UL TBs	M		155	KPM7	5.1.1.7.6 [23]
8	Total num. of UL TBs	M		157	KPM8	5.1.1.7.8 [23]
9	Raw DL scheduling info	M	Real-time telemetry without introducing new service model	137	RAW1	-
10	Raw UL scheduling info	M		150	RAW2	-
11	Interference detection	I	Detection with 40x less telemetry bandwidth by sampling and correlation at source (vRAN)	265	ID	-
12	ARIMA	I	Up to 12% increase in cell throughput by real-time inference of user signal quality	81	ML1	[44]
13	Decision tree	I	Reclaim up to 70% CPU cycles from vRAN PHY by real-time inference of PHY processing runtime	10495	ML2	[56]
14	Random forest	I	Up to 30% reduction in user RTT latency by real-time prediction of user buffer status report	51	ML3	[94, 111]
15	Earliest deadline first slicing	C	Near zero delay budget violation for URLLC slices compared to conventional scheduler	174	SL1	[64]
16	Static slicing	C	Up to 21% improvement in cell throughput for eMBB slices compared to conventional scheduler	41	SL2	[83]
17	Proportional fair slicing	C	Up to 25% improvement in cell throughput for eMBB slices compared to conventional scheduler	384	SL3	[83]
18	Inter-Cell Interference Coordination	C	Up to 30% increase in cell throughput by real-time coordinated radio resource scheduling	73	ICIC	[54]

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

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, as well as the collection frequency and granularity (e.g., raw IQ samples vs average energy per resource block). As we show in §7.2, performing this pre-proceessing inline, 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 (e.g., localization [74, 98] and channel estimation [48, 49]).

Furthermore, many RAN control loops can benefit from real-time parameter prediction to improve the network performance [39, 40, 51, 58, 77, 95, 102, 111, 113]. Due to the O-RAN RIC latency, xApps provide predictions that are 10s of milliseconds old [33, 88]. Using Janus, we were able to build codelets that *perform inference inside the vRAN functions with under* 10*ms latency*. We demonstrate this with the inference models listed in lines 12-14 of Table 3. The first is an ARIMA model for the prediction of user signal quality, following a methodology similar to [44]. The second is a quantile decision tree for the prediction of signal processing task runtimes, using the methodology in [56]. More complex models, such as the Random Forest in [111], are more difficult to implement with eBPF, as they result in a large number of bytecode instructions (> 100K) making the verification process challenging. 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 is similar to the serialization feature offered by frameworks like Tensorflow for micro-controllers [104] and could be extended to other commonly used ML models (e.g., LSTM).

Flexible and real-time control: Many enterprise applications require network slicing for service QoS guarantees [50, 55] . Existing O-RAN service models allow for a set of pre-defined slice scheduling policies [45, 72, 97] 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 MAC 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, buffer sizes, signal quality etc), along with a structure carrying the scheduling decision. Using this hook, we were able to implement three slicing schedulers as Janus codelets, as listed in Table 3 (lines 15-17).

Custom control commands also allow for the flexible deployment of *near real-time* control applications, without any further modifications in the vRAN functions for the introduction of new service models. For example, by using a custom control command that allowed us to modify the number of allocated resource blocks of each cell centrally from the Janus controller, we were able to implement an inter-cell interference mitigation application [54] (line 18 in Table 3). The application introduces Almost Blank Subframes to cells in a centralized manner, ensuring that neighboring cells do not transmit/receive data at the same time.

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, 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. [47, 107]. 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 in 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 μs vs 2.4 μs for the codelet of Listing 3). This is because the helper function incurs a higher overhead compared to the multiplication and addition instructions of the other codelet, indicating that the maximum number of instructions 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 μs)

```
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 μs)

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 the codelet if a threshold is exceeded. A helper function (`mark_init_time()`) is added at the beginning of the codelet (Fig 3), which stores the current time in a thread local variable. The patcher introduces checkpoints in selected locations (Algorithm 1) that invoke a helper function (`runtime_limit_exceeded()`, line 18 in Fig 3), which checks the elapsed runtime 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

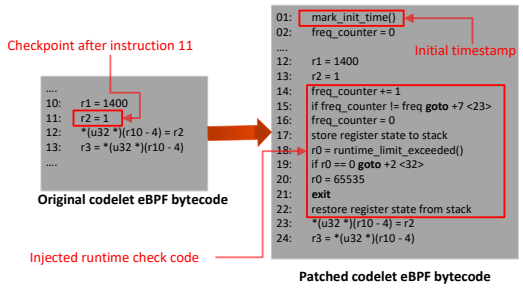


Figure 3: Simplified version of code patching process.

Fig 3). The runtime threshold is specified during the loading of the codeletset. Finally, the patcher updates all jump offsets to account for the injected instructions. This approach allows us to verify the patched bytecode, 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 Algorithm 1. Initially, Janus adds checkpoints right after the invocation of helper functions marked by the vendors as *long lasting* (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 and 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.

For finer control, Janus allows vendors to instrument checkpoints in their helper functions, which perform a similar operation as the patch of Fig. 3. For example, in the case of the

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

$C \leftarrow \{\}$;
foreach *instruction* f *in* F **do** $C \leftarrow C + f$;
foreach p *in* P **do**
  ins $\leftarrow 0$;
  fins $\leftarrow$ first instruction of p;
  foreach *instruction* i *in* p **do**
    ins $\leftarrow$ ins $+ 1$;
    if i *has already checkpoint* **then**
      ins $\leftarrow 0$;
    else
      if $ins = N$ **then**
        $C \leftarrow C + i$;
        ins $\leftarrow 0$;
      end
    end
  end
  if p *is cycle and no checkpoint was added* **then**
    $C \leftarrow C +$ fins;
  end
end

random forest model (see §4), we added such checks after the inference of each estimator (tree) of the model. Given that the overhead can become significant even for a few checkpoints (e.g., in codelets with tight loops), 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 instruction. This guarantees that a check is performed at least once every $M \times N$ instructions.

Pre-empted control loops: Each control hook must provide a default control action, in case a control codelet is pre-empted (or fails). 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).[2] For example, in the case of the MAC scheduling hook of Table 2, the default action could be to assign the radio resource blocks equally. Similarly, for the xRAN packet transmission hook, it would be to forward any xRAN packet. If a codelet is pre-empted N times (configurable parameter), then the codeletset it belongs to is unloaded and the Janus controller is notified. 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

[2]The same mechanisms address the case of no codelet being loaded.

5.2 Flexible and verifiable IO schemas

Janus allows the definition of arbitrary output and input control schemas, loaded with the codelet at runtime. Both output and input control data are serialized by Janus 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 and cannot read arbitrarily large control inputs, which could violate memory safety. Protobuf messages are defined by the developer and can contain variable size fields (e.g. repeated fields). 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 input and output 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 related specifically to output schemas and making sure that an incorrectly formatted message cannot lead to memory violations. Consider a case where a programmer allocates 30B of memory, casts it as an `Example` message, sets the number of elements to be 16 and calls the output helper function to send the data to the controller. 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 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 above are still functionally incorrect and send garbled data, but do not violate safety.

6 IMPLEMENTATION DETAILS

6.1 Janus components implementation

Janus device: The Janus device is implemented in C as a dynamically linked library. It is based on a uBPF [71], which we extended to add support for the Janus maps, helper functions, the mechanism for input/output APIs etc. Overall, we had to add ~7K lines of code to the basic implementation. The Janus device was developed without making any assumptions about the threading model of the vRAN functions (e.g., affinity of threads calling hooks). However, Janus can be configured to

allow the 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.

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. For the codelet verification, we used the open source PREVAIL verifier [60], which we extended with ~1K lines of code to add support for Janus specific functionalities (i.e., helper functions). Finally, the Janus patcher relies on pyelftools [1] and LLVM [16] 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 uBPF [71] for the conversion of the bytecode to x86 JIT code and on nanopb [9] for the compilation of protobuf messages for the codelets' output schemas.

6.2 Thread safety of Janus codelets

Janus hooks can be invoked by multiple threads, which could raise concurrency issues. We have thus implemented a number of mechanisms that can help with the thread-safety of Janus:

Thread-safe maps – Janus provides a lock-free, thread safe hashmap implementation, as well as thread-local hashmaps and arrays, which can be useful for codelets that only want to store local state. It also comes with a wait-free thread-safe ringbuffer implementation, that is used for IO.

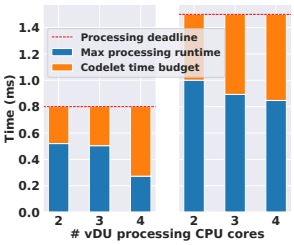
Atomic operations – eBPF bytecode already provides support for atomic operations (e.g., `atomic[64]_[fetch_]add`), which we have ported to the Janus x86 JIT compiler.

With regards to the thread-safety of the hook context, it is important to note that codelets are executed inline, so they are implicitly granted access to the same RAN state as their host code. As such, they enjoy the same multi-threading protection as any other inlined function call that might be using this state. This could come in the form of a hook being called with different context state for different thread instances or with the vendor passing a unique thread identifier as part of the context, to allow the codelet to differentiate calling instances.

6.3 vRAN integration

Integrating Janus to vRAN functions is simple and fast. As a proof-of-concept, we integrated Janus devices to two vRAN software implementations. One is the commercial-grade 5G vCU/vDU implementation developed by CapGemini [46], and based on Intel's FlexRAN PHY design [69]. The other is the open source OAI [37]. 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. The initial integration effort took 2 weeks for the CapGemini stack, with a single developer, who was unfamiliar with our codebase. The OAI integration effort was similar.

(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

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 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 [93] 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 [56, 70, 106].

For the evaluation, we use three setups. The first is an end-to-end setup, composed of the commercial grade 5G vRAN stack of CapGemini [46], with integrated Janus devices (see §6), a commercial-grade 5G core, 100MHz 4×4 Foxconn radio units and 5G OnePlus Nord smartphones (Fig 4a). Aside from LDPC [67], all vRAN tasks run on x86 processors. Using this setup, we are able to deploy 3 cells with a 4×4 MIMO configuration[3] and to generate a max of 1Gbps downlink and 45Mbps uplink per cell, which is a representative capacity and MIMO configuration of a commercial deployment. This setup is instrumented with Janus collecting IQ samples, FAPI and RLC data (see Table 1). In the remainder of this section, and when referring to this setup, we present our measurements for a single cell for simplicity, given that we obtained similar results for all 3 cells. 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 uses a Janus emulator, which is a single-threaded process that

[3]We do not investigate massive MIMO or mmWaves, due to lack of access to such an implementation. We plan to investigate this in our future work.

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 invoked by affinitized threads and scheduled using the `SCHED_FIFO` policy, with a scheduling priority of 94. For runtime measurements, we use a time measuring framework based on the guidelines in [85]. Finally, note that Janus requires a single CPU core for all of its functionalities. The rest are allocated to the vRAN functions.

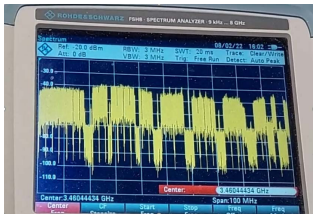
Codelet runtime budgets: We use our 5G RAN setup to determine how much time we can allocate to Janus codelets without affecting the RAN performance. We focus on the PHY, as all other layers have less stringent timing requirements. Transmissions and receptions of packets in the PHY occur in Transmission Time Intervals (TTIs) of a fixed duration [56]. Using our CapGemini setup we measure the runtimes of the PHY per TTI when saturated (1Gbps DL and 45Mbps UL) over a period of 15 minutes. Fig 4b shows the maximum UL and DL runtimes as a function of CPU cores. We also plot the processing deadline for the given configuration based on the vendor guidelines. The difference (orange), is the runtime budget for Janus codelets, and it varies from 200 μs to 600 μs.

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 20μs.

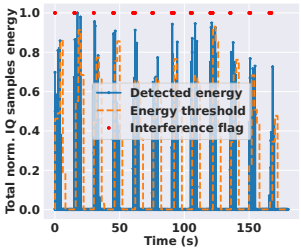
7.2 End-to-end system evaluation

Here we demonstrate the flexibility of collecting telemetry using Janus codeletsets and the safety provided by Janus when loading the codelets in an end-to-end environment. For the safety part, we evaluate 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 [60]. Similarly, we point the reader to the references and the short description 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.



(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.

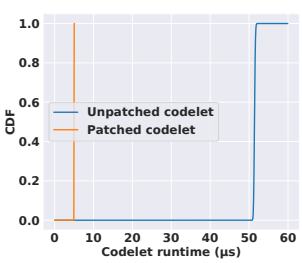
While running the measurement, we load the Janus codelets for interference detection, described in §4. We implement a simple interference detector at the controller that continuously tracks the mean and the variance of the input signals per resource block and declares interference if the input is larger 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.

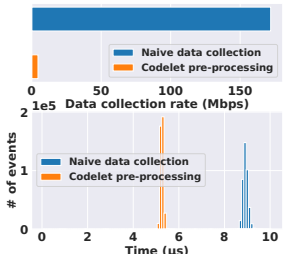
Our deliberately inefficient codelet is verified as correct by the verifier, as it is deemed safe in terms of memory access, as well as provably terminates (bounded loop). However, once this codelet is loaded to our end-to-end vRAN deployment unpatched, it crashes the vRAN. As it can be seen by the CDF in Fig 6a, this codelet runs for 51.4 μs on the median and 52.2 μs 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 μs on average, which is greater than the 600 μs 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 μs. 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 μs and the 99.999 percentile becomes 5.04 μs. 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



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



(b) Rate of data collection and runtime histogram of codelet-sets for interference detection use case.

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

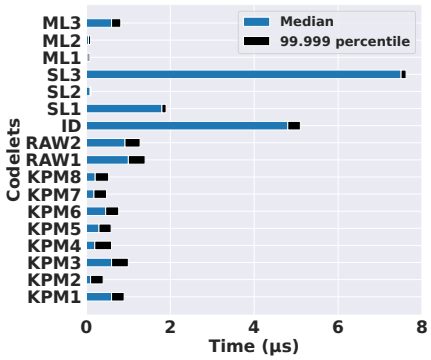


Figure 7: Patched codelets (Table 3) worst-case runtime.

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 output data throughput to the Janus collector, as well as the runtime of the two codelet-sets, as is illustrated in Fig 6b. As shown at the top sub-plot, the naive interference detection method results in a data collection rate of 172Mbps, while the inline pre-processing method results in a collection rate of 4.5Mbps; almost 40× less. The runtime of the codelets with pre-processing is lower by ∼ 3.5 μs 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.

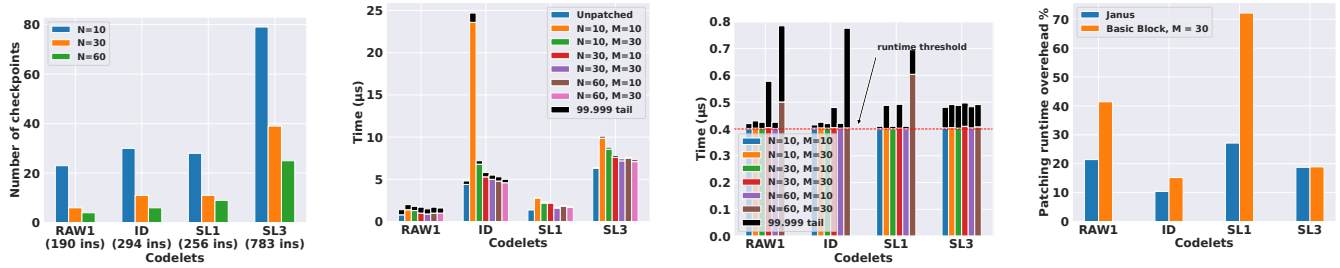
Codelets runtimes: Due to lack of space we don't discuss each scenario from §4 in depth. Instead, we report the median and tail (99.999) runtimes of the 18 codelets of Table 3 in Fig 7, using the shorthand names of the table. We consider the worst execution case for each codelet (i.e., maximum number of devices, maximum bandwidth etc) and report the time for patched codelets. We use a patching distance $N = 60$ 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 20μs 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). To put this into perspective, all codelets combined take $< 0.5\%$ of the eMBB slot processing time budget. 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 (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



(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 0.4μs 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.

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.

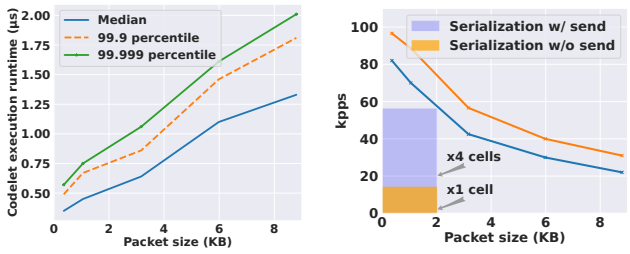
Finally, we compare the checkpoint method of Algorithm 1, with an alternative method proposed in [108, 109], where checkpoints are introduced on each basic block of the control flow graph of the patched code. For the basic blocks method, we use a sampling frequency of $M = 30$, which yields similar pre-emption 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.

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 placing data to the output map (this

	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.



(a) Codelet execution runtime

(b) Max kpps

Figure 9: Codelet execution runtime and maximum kpps for `SizeMessage1` message [90] with fields of varying size.

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 [90]. 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 μs and then gradually increases for large packet sizes, but always remains below 2μs 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 networking overhead for serializing and sending telemetry to the controller. We use the same setup as in the previous experiment and we

measure the maximum achievable packet sending rate (blue line in Fig 9b). For small packets ($< 500B$), Janus can send more than 80kpps, which drops to 20kpps when sending 9KB packets. To put this into perspective, the number and size of packets for a single cell falls within the orange area of Fig 9b for the telemetry codelets of §4. This means that a single Janus device can handle up to 4 cells (light blue and orange area) for demanding use cases and more than double for more lightweight monitoring. The output rate of Janus is currently limited by the socket-based UDP communication. This can be seen by the orange line of Fig 9b, where we discard packets instead of sending them out, for a 25% increase in the output rate. In the future, we plan on introducing kernel bypassing for the networking (e.g., DPDK) to alleviate this limitation.

8 DISCUSSION AND LIMITATIONS

Support for complex ML models - As discussed in §4, the development of large ML-models (e.g., Random Forests) as Janus codelets is challenging.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 several RAN ML use cases rely on the same models (e.g., Random Forests, LSTMs and RNNs [28, 51, 66, 77, 96, 102, 110, 111]). We are planning on extending this approach to support additional widely-used ML models beyond Random Forests.

Janus in vRANs with HW accelerators - In this work, we focused on vRANs that only use look-aside hardware accelerators for offloading computationally heavy tasks, like LDPC decoding (e.g., Intel FlexRAN). In the case of vRANs that rely on inline accelerators for the full processing of the physical layer (e.g., Nvidia Aerial [10], Qualcomm [14] and Marvell [12]), Janus cannot be integrated to the physical layer. However, we believe that it can still be very useful for vRAN monitoring and control, considering that the remaining vRAN layers are implemented in software. In fact, 6 out of the total of 8 hooks of Tables 1 and 2 are placed at the higher layers and, as such, are applicable to any vRAN implementation.

Support for architectures beyond x86 - Janus currently supports x86-based vRAN functions. However, given the emergence of other vRAN architectures (e.g. ARM-based [2]), we are planning on extending Janus to support such use cases. This mainly requires modifications to the JIT compiler of Janus, that converts the eBPF bytecode to native code.

Other use cases for Janus - Here, we presented a number of use cases that could benefit from the dynamic service model capabilities of Janus. As a future work, we are planning on exploring additional use cases, including localization [74], channel estimation [77, 80], smart scheduling of radio resources for energy savings [87, 100], resilience (e.g., in the spirit of [58] and anomaly detection [57, 86]. We will also explore the use of Janus in other domains, like for example for telemetry and control applications for mobile core functions, like the UPF.

9 RELATED WORK

Pushing arbitrary code to vRAN functions - The works in [54, 82] 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, making their proposal unacceptable for realistic deployments. They are also only shown to operate on lower-end setups (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 explored as a way of improving fault tolerance by periodically saving the software state [78, 79, 91, 112, 114]. 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 those of Janus. Closer to Janus, the works in [108, 109] 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 - RAN data collection has been explored both in terms of API specifications (e.g., O-RAN RIC E2 service model [35, 97] and FlexRAN API [54]) to logging systems (e.g. OAI T-tracer [11], SCOPE data collection module [41])). 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 eBPF-based data collection solutions, which either offer a fixed set of metrics (e.g. Hubble [7]) or data can only be exported in certain formats, like counters and histograms (e.g., ebpf_exporter [6]).

10 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 18 applications in 3 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.

ACKNOWLEDGMENTS

This work was partially funded through the Open Networks Programme within the UK Department for Science, Innovation and Technology.

REFERENCES

[1] 2021. pyelftools. https://github.com/eliben/pyelftools.
[2] 2022. Arm-based Private 5G Network-in-a-box with MEC Support. https://www.virtualexhibition.o-ran.org/classic/generation/2022/category/open-ran-demonstrations/sub/open-interface/149.
[3] 2022. Awesome eBPF. https://github.com/zoidbergwill/awesome-ebpf.
[4] 2022. Cilium – eBPF-based Networking, Observability, Security. https://cilium.io/.
[5] 2022. eBPF. https://ebpf.io/.
[6] 2022. ebpf_exporter. https://github.com/cloudflare/ebpf_exporter.
[7] 2022. Hubble exported metrics. https://docs.cilium.io/en/stable/operations/metrics/#hubble-exported-metrics.
[8] 2022. Katran – a scalable network load balancer. https://github.com/facebookincubator/katran.
[9] 2022. nanopb. https://github.com/nanopb/nanopb.
[10] 2022. Nvidia Aerial SDK. https://developer.nvidia.com/aerial-sdk.
[11] 2022. OAI T Tracer. https://gitlab.eurecom.fr/oai/openairinterface5g/-/wikis/T.
[12] 2022. OCTEON Fusion CNF95xx Press Briefing Presentation. https://www.marvell.com/content/dam/marvell/en/company/media-kit/octeon/marvell-octeon-fusion-launch-press-deck.pdf.
[13] 2022. Open source Kubernetes observability for developers. https://px.dev/.
[14] 2022. Qualcomm 5G RAN Platforms: Driving for Full-Scale Open vRAN Commercialization. https://www.qualcomm.com/news/onq/2022/10/qualcomm-5g-ran-platforms--driving-for-full-scale-open-vran-comm.
[15] 2022. The Falco Project – Cloud-Native Runtime Security. https://falco.org/.
[16] 2022. The LLVM compiler infrastructure. https://llvm.org/.
[17] 2023. eBPF Documentation. https://ebpf.io/what-is-ebpf/.
[18] 2023. eBPF Linux kenrel Documentation. https://www.kernel.org/doc/html/latest/bpf/index.html.
[19] O-RAN Working Group 3. 2021. Use Cases and Requirements. *O-RAN.WG3.UCR-v01.00* (2021).
[20] 3GPP. 2018. 3GPP TS 38.410: NG general aspects and principles. (2018).
[21] 3GPP. 2018. 3GPP TS 38.463: E1 Application protocol (E1AP). (2018).
[22] 3GPP. 2019. 3GPP TS 38.470: F1 general aspects and principles. (2019).
[23] 3GPP. 2020. 3GPP TS 28.552: Management and orchestration: 5G performance measurements. (2020).
[24] 3GPP. 2020. 3GPP TS 32.425: Performance Management (PM); Performance measurements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN). (2020).
[25] 3GPP. 2020. 3GPP TS 38.322: Radio Link Control (RLC) protocol specification. (2020).
[26] 3GPP. 2020. 3GPP TS 38.331: Radio Resource Control (RRC) protocol specification . (2020).
[27] 3GPP. 2020. 3GPP TS 38.420: Xn general aspects and principles. (2020).
[28] 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.
[29] ORAN Alliance. 2019. O-RAN WhitePaper-Building the Next Generation RAN. *O-RAN Alliance, Tech. Rep., Oct* (2019).
[30] ORAN Alliance. 2020. O-RAN use cases and deployment scenarios. *White Paper, Feb* (2020).
[31] 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).
[32] ORAN Alliance. 2021. O-RAN Fronthaul Control User and Synchronization Plane Specification v7.0.
[33] 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).
[34] ORAN Alliance. 2022. O-RAN E2 Service Model (E2SM), RAN Control 1.03. *O-RAN.WG3.E2SM-RC-v01.03* (2022).
[35] O-RAN Alliance. 2021. O-RAN E2 Application Protocol (E2AP) v2.0. *ORAN-WG3.E2AP-KPM-v02.00* (2021).
[36] O-RAN Alliance. 2021. O-RAN Minimum Viable Plan and Acceleration towards Commercialization. *White Paper, June* (2021).
[37] OpenAir Software Alliance. 2022. Open Air Interface Project. https://openairinterface.org/about-us/.
[38] 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.
[39] 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.
[40] 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.
[41] 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.
[42] 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.
[43] 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).
[44] 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.
[45] Cambridge Consultants. 2022. Wireless breakthrough for the Ocado Smart Platform. https://www.cambridgeconsultants.com/case-studies/wireless-breakthrough-ocado-smart-platform.
[46] CapGemini Engineering. 2022. 5G gNodeB. https://capgemini-engineering.com/nl/en/services/next-core/wireless-frameworks/.
[47] 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.
[48] 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/.

[49] 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.

[50] 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.

[51] Capgemini Engineering. 2021. Intelligent 5G L2 MAC Scheduler. *White Paper, Feb* (2021).

[52] Small Cell Forum. 2021. 5G FAPI: PHY API Specification.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] Xenofon Foukas, Fragkiskos Sardis, Fox Foster, Mahesh K Marina, Maria A Lema, and Mischa Dohler. 2018. Experience building a prototype 5G testbed. In *Proceedings of the Workshop on Experimentation and Measurements in 5G*. 13–18.

[58] 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.

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

[60] 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.

[61] 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.

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

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

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

[65] 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.

[66] 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.

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

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

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

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

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

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

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

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

[86] Georgios Patounas, Xenofon Foukas, Ahmed Elmokashfi, and Mahesh K Marina. 2020. Characterization and Identification of Cloudified Mobile Network Performance Bottlenecks. *IEEE Transactions on Network and Service Management* 17, 4 (2020), 2567–2583.

[87] Ujjwal Pawar, Bheemarjuna Reddy Tamma, and Franklin A Antony. 2021. Traffic-Aware Compute Resource Tuning for Energy Efficient Cloud RANs. In *2021 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 01–06.

[88] 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).

[89] 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.

[90] Protocol Buffers. 2022. Protobuf benchmark proto file. https://github.com/protocolbuffers/protobuf/blob/fb77cc9d9f066a8ce4f12e8d5f76188d48101444/benchmarks/google_size.proto.

[91] 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.

[92] Raymond Knopp. 2022. OAI Layer 2 Protocol Stack. https://www.openairinterface.org/docs/workshop/1stOAINorthAmericaWorkshop/Training/KNOPP-OAI-L2.pdf.

[93] 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.

[94] Flavien Ronteix-Jacquet, Xavier Lagrange, Isabelle Hamchaoui, and Alexandre Ferrieux. 2022. Rethinking Buffer Status Estimation to Improve Radio Resource Utilization in Cellular Networks. In *2022 IEEE 95th Vehicular Technology Conference:(VTC2022-Spring)*. IEEE, 1–5.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

[99] 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.

[100] Rajkarn Singh, Cengis Hasan, Xenofon Foukas, Marco Fiore, Mahesh K Marina, and Yue Wang. 2021. Energy-efficient orchestration of metro-scale 5G radio access networks. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 1–10.

[101] Small Cell Forum. 2022. 5G FAPI: PHY API Specification. http://scf.io/en/documents/222_5G_FAPI_PHY_API_Specification.php.

[102] 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.

[103] Telefonica. 2021. Telefonica views on the design, architecture, and technology of 4G/5G Open RAN networks. *White Paper, Jan* (2021).

[104] TensorFlow. 2022. TensorFlow Lite for Microcontrollers. https://www.tensorflow.org/lite/microcontrollers.

[105] 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.

[106] OpenAirInterface Wiki. 2022. OpenAirKernelMainSetup. https://gitlab.eurecom.fr/oai/openairinterface5g/-/wikis/OpenAirKernelMainSetup.

[107] 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.

[108] 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.

[109] 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.

[110] 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).

[111] 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.

[112] 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.

[113] 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.

[114] Avi Ziv and Jehoshua Bruck. 1997. An on-line algorithm for checkpoint placement. *IEEE Transactions on computers* 46, 9 (1997), 976–985.