Snout: A Middleware Platform for Software-Defined Radios

Johannes K Becker, *Student Member, IEEE,* David Starobinski, *Senior Member, IEEE.*

***Abstract*—The plethora of Internet of Things (IoT) protocols and the upcoming availability of new spectrum bands for wirelessly connected devices have made software-defined radio (SDR) technology increasingly useful to interact with radio-based communication. While SDR-based tools have grown in popularity in recent years due to their flexibility and adaptability towards new protocols, SDR software interfaces remain highly complex and technical, and inherently require specialized skillsets in digital signal processing (DSP) to operate. To address this problem, we present Snout, an SDR middleware platform that encapsulates and abstracts much of the current complexity in SDR toolchains. This allows SDR developers to create wireless networking applications usable by a wide range of users. For instance, network security professionals can monitor the IoT landscape across multiple protocols without needing to interact with the underlying software-defined DSP. Snout implements interfaces with common network analysis tools to allow for integration with traditional network security solutions, facilitating use cases such as traffic analysis or rogue device detection. Its software architecture enables scalability in terms of protocols and processing by modularizing the signal processing pipeline. To demonstrate Snout's capabilities, we show how it can encapsulate GNU Radio flowgraphs, facilitate simultaneous multi-protocol scanning, and convert existing SDR-based protocol implementations into fully contained applications. We further demonstrate how Snout can handle GNU Radio flowgraphs with other signal processing software simultaneously. Through extensive experiments, we demonstrate that Snout incurs limited CPU performance overhead below 4% and a memory footprint below 100MB, and handles large amounts of events with sub-millisecond latency.**

***Index Terms*—software-defined radio, GNU Radio, middleware, wireless devices**

I. Introduction

Software-defined radio (SDR) technology has become increasingly accessible from both cost and signal processing perspectives, thanks to open-source, community-based SDR frameworks, such as GNU Radio [1], [2], and a large ecosystem of hardware platforms (such as RTL-SDR [3], Ettus USRP [4], HackRF One [5], LimeSDR [6], and BladeRF [7]). However, end users of a typical SDR application need to be skilled in digital signal processing (DSP) to be able to process raw signal data.

While GNU Radio is the most popular open-source SDR framework, it is notoriously complicated and hard to develop for [8, p. 59]. GNU Radio provides a block-based "flowgraph" model for interacting with signals received and transmitted by SDR software, which requires the end user to be able to understand and manage flowgraph blocks in the low-level signal processing pipeline, such as managing bitrate and bandwidth settings or configuring discrete filters and signal demodulation algorithms, which are skillsets that are hard and span a huge number of areas [9]. This limits its use to users with advanced, niche technical skills, and prevents large groups of potential users who could benefit from an SDR-based toolbox, but lack the technical experience, from using it.

There is a lack of middleware that abstracts the low-level complexity of signal processing and SDR-specific knowledge away from the user, while still allowing the user to configure the underlying functionality in a more user-centric way, e.g., by selecting a Zigbee channel on which to scan for devices, rather than to require detailed knowledge of the technical implementation of the flowgraph that handles the protocol.

In this work, we propose a middleware framework, coined *Snout*, to bridge this gap. Snout provides an abstraction layer decoupling user interaction from low-level receivers and transmitters implemented in GNU Radio flowgraphs or other signal processing code. Snout provides a simple description language that lets users configure and operate SDR-based scanning capabilities without the need for deep signal processing knowledge. Snout further allows for composability of flowgraph-based capabilities, i.e. executing receivers for multiple communication protocols in parallel, which is usually non-trivial and requires advanced knowledge in both signal processing and proficiency in the GNU Radio flowgraph editor. As a result, Snout allows SDR developers to create applications that are usable by non-experts, and that interface with non-SDR software in a flexible way, while still benefiting from the advantages of the underlying SDR platform. Ensuring that the performance of the underlying, computationally intensive system is not impacted by such a middleware layer is a key challenge, which our work addresses and validates via extensive testing.

The contributions of this work are as follows:

1) We present a middleware architecture, Snout, that encapsulates the complexity of GNU Radio.
2) We develop an extensible interface that allows for third party plugins to extend functionality.
3) We introduce a configuration language for control and instrumentation of SDRs via Snout.
4) We develop a standardized communication architecture that allows data exchange between Snout, the underlying SDR platform, and other systems (such as databases or

J Becker and D Starobinski are with the Department of Electrical and Computer Engineering at Boston University, Boston, MA.
Contact: {jkbecker,staro}@bu.edu

Fig. 1. A typical SDR setup consisting of a host computer, an SDR device connected via USB, communicating with wireless devices (adapted from [10]).

third-party analysis tools) via established communication libraries.

5) We introduce four case studies that showcase applications of Snout for popular IoT protocols.
6) We thoroughly evaluate and characterize the performance overhead of Snout in terms of CPU and memory usage compared to running bare low-level signal processing code, and benchmark the performance of the underlying message-passing system.

The rest of this paper is structured as follows: Section II introduces existing software radio frameworks and expands on the underlying technologies that are most relevant to Snout. Section III discusses related works. Section IV describes the middleware architecture and its design considerations. Section V introduces case studies to demonstrate Snout and measure its performance. Section VI summarizes our findings and discusses avenues for future work.

II. Background

This section introduces general software-defined radio terminology, surveys major software radio frameworks, expands on the GNU Radio framework in particular, and explains the basics of ZeroMQ-based messaging.

A. Software-defined radio terminology

While software-defined radio systems can be implemented using a plethora of different hardware and software platforms, there are generic components that make up a typical SDR setup in the way it is presumed in this work. A typical setup consists of a host computer and one or more SDR devices, or radios.

The host computer runs hardware-specific drivers to communicate with its radio, typically via USB or Ethernet (see Figure 1). These drivers may be proprietary or open-source, and may apply to a range of radios that share a common architecture, or be specific to one particular type of radio.

SDR devices consist of a radio front-end (i.e. the circuitry handling the analog radio frequency (RF) signals received by and transmitted via its antennae), analog-to-digital (and vice versa) converters (ADC/DAC), and a variable number of additional digital signal processing components. SDR hardware [3]–[7] is available in a wide range of capabilities, and is typically characterized by its frequency range, the number of transceivers, maximum instantaneous bandwidth (i.e., the widest-possible RF signal it can capture from the spectrum), the sample size (i.e., the bit-resolution of the ADC/DAC), and their communication interface to the host computer, as well as additional optional features such as field-programmable gate array (FPGA) chips.

In a simple configuration, most of the digital signal processing for handling arbitrary radio protocols is developed in software, which is executed on the host computer, in which case the data transmitted between the SDR and the host computer is a stream of quadrature (complex) signal samples. In more capable (and expensive) systems, parts of the signal processing can be implemented in the SDR's FPGA chip, which offloads computational load from the host computer's CPU and reduces the load on the data connection between the host and the SDR. This enables more advanced and high-throughput protocols, such as Wi-Fi [11], LTE [12], and 5G [13].

The Snout middleware executes on the host computer and supports a large number of SDR platforms, as discussed in Section IV-A, and crucially, does not limit or interfere with the performance considerations and design trade-offs discussed above.

B. Software radio ecosystem

a) Software radio frameworks.: There exist several popular software radio frameworks that let users develop any kind of digital signal processing capabilities and/or compose new functionality from existing building blocks. The MathWorks Communication Toolbox is a commercial solution for "analysis, design, end-to-end simulation, and verification of communications systems" [14]. It supports numerous wireless protocols and a range of common SDR platforms. Similarly, National Instruments' LabVIEW SDR Lab [15] is a powerful SDR-based development framework for USRP (Universal Software Radio Peripheral) SDR devices. However, the costs associated with their commercial licensing makes these frameworks inaccessible for a large number of potential users of SDR technology. In contrast, SDR frameworks such as GNU Radio [1], [2], [16], Pothosware [17] and the underlying SoapySDR framework [18], RedHawk SDR [19] or LuaRadio [20] enable building powerful SDR applications based on *open-source* code.

While each of these frameworks has slightly different hardware support and functionality, they all share a common model of using composable data flowgraphs to model signal processing functionality, and then compiling the flowgraph representation into some executable code that can be run with a number of SDR hardware platforms.

b) General-purpose DSP-based SDR software: typically implements a "waterfall-style" view of the raw signal data in a graphical window, accompanied by a range of tools for filtering, demodulating, analyzing, and processing the raw signal, either providing standalone implementations (e.g., QIRX SDR [21]) or building on one of the aforementioned SDR frameworks such as SoapySDR (e.g., CubicSDR [22], SigDigger [23], and OpenWebRX [24]) or GNU Radio (e.g., Gqrx SDR [25], ShinySDR [26], and Project sdrangelove [27]). Some tools, such as SDRangel and OpenWebRX, offer web-based user interfaces, which allow for distributed and remote interaction with SDR data sources [24], [28].

While many of these tools use one of the aforementioned SDR frameworks to achieve compatibility with a range of hardware platforms, other tools such as inspectrum [29] focus entirely on signal analysis and only take raw sample files as input, avoiding any direct dependency on SDR hardware altogether, but sacrificing any real-time capabilities.

c) Special-purpose SDR software: exists for multiple applications dealing with RF technology. Tools such as SodiraSDR [30], Linrad [31], Studio1 [32], and HDSDR [33] focus on AM/FM decoding and other Ham radio-specific functionality. QRadioLink [34] (based on GNU Radio) additionally supports VoIP functionality. These tools have highly technical interfaces that are aimed at amateur radio enthusiasts.

SDR-based wireless security monitoring software solutions such as SCEPTRE [35] or Skylight [36] often go far beyond simple signal decoding and offer feature detection, raw signal playback, supporting multi-antenna setups and advanced database integration and monitoring of multiple wireless protocols for surveillance purposes. While these tools are extremely powerful, they are built on an opaque technology stack. They are also inaccessible to large parts of the wireless community due to their surveillance and defense-focused licensing. Even more so than amateur-oriented SDR software, these tools are geared towards highly skilled users with deep signal processing and communications intelligence expertise.

C. GNU Radio

Of the aforementioned software radio frameworks, GNU Radio [1], [2] is by far the platform with the largest developer and user community. Its user interface and top-layer code are implemented in Python, whereas the signal processing is implemented in C++, which is made accessible to the Python code via SWIG (until GNU Radio 3.8) and PyBind11 (since GNU Radio 3.9). GNU Radio allows users to develop and install so-called out-of-tree (OOT) modules to extend its functionality in arbitrary ways, and currently lists over 300 community-developed modules as installable "recipes" [37], [38]. GNU Radio is compatible with a large range of SDR devices using OOT modules such as `gr-uhd` [39], `gr-osmosdr`, `gr-iio`, `gr-soapy`, and `gr-limesdr`[1].

a) Anatomy of a GNU Radio flowgraph: As of GNU Radio 3.8, flowgraph files are formatted in YAML [41], and contain a textual description of all contained variables, parameters, blocks as well as their connections. These files are typically generated and edited using the *GNU Radio Companion* flowgraph editor, and compiled to native Python code via the *GNU Radio Companion Compiler* (`grcc`). The resulting application can be run from within GNU Radio Companion or executed as a stand-alone Python application.

Figure 2 shows the visual representation of a simple GNU Radio flowgraph in GNU Radio Companion. In this flowgraph, Bluetooth LE advertising signal data on channel 37 (at 2402MHz) is received from a USRP source block, processed through multiple subsequent blocks until the resulting bytestream of demodulated raw packet data is passed into a ZeroMQ PUSH sink (see also Section IV-C).

[1]These modules are available via the list of official GNU Radio recipes [37].

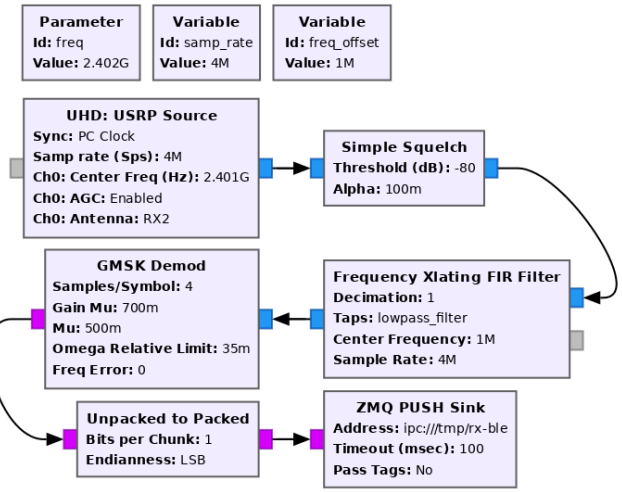

Fig. 2. A simple GNU Radio flowgraph for receiving a Bluetooth LE (BLE) signal (adapted from [40]).

GNU Radio flowgraphs can either be compiled to Python code as a stand-alone application, or to a so-called *hierarchical block*. This type of block does not compile to an application, but instead compiles to a block that is available inside GNU Radio Companion to connect with other components. This allows for encapsulation of complex low-level processing into sub-blocks to improve manageability of the overall flowgraph, such as in the WiFi transceiver module of Bloessl et al. [42].

In addition to data processing blocks, which can be connected to each other, Parameter and Variable blocks (see top of Figure 2) can be used to store reused values in a central place, or calculate flowgraph block parameters based on formulae that depend on them. The key difference between the two is that Parameters can be passed to the application as command-line arguments, whereas variables are internal to the flowgraph.

b) Runtime communication with a GNU Radio flowgraph: At runtime, interaction with a flowgraph is possible in multiple ways. If the flowgraphs contain graphical user interface (GUI) elements, such as sliders or numerical input fields, their values will update the flowgraph in real-time. When a flowgraph is run in command-line mode, variables can be read out and changed using an XMLRPC Server[2] or ControlPort [43].

If more direct control over a flowgraph is desired, compiled flowgraphs can be imported into other applications as regular Python modules. Then, all methods for running and stopping, as well as getter and setter methods for any variables or parameters in the flowgraph are directly accessible via the top-block class representing the flowgraph as a whole.

III. Related Works

In 2009, Li et al. [44] proposed an architecture for secure software-defined radio, which comprises a secure radio middleware to protect SDR applications from unwanted reconfig-

[2]The XMLRPC Server is part of the core GNU Radio installation, available in the official repository [16] at `gr-blocks/grc/xmlrpc_server.block.yml`.

uration and the execution of malicious code. While this architecture is also based on GNU Radio, it otherwise builds on an architecture that leverages virtual machine encapsulation and focuses on enforcing certain security policies when executing SDR code. While Snout can be used in the context of security research in the context of wireless device security, it does not focus on protecting the SDR code from a host-based threat, and does not require virtualization-based encapsulation of its runtime.

Shome et al. [45] presented a Software-Defined Networking-oriented (SDN) framework for controlling SDR-based hardware using a control plane and data plane architecture and the OpenFlow protocol. While this framework focuses on making SDR hardware interoperable with SDN principles, our work focuses on the abstraction of protocol-specific tools as well as multi-protocol scanning using SDR, which are not covered in the work of Shome et al.

In 2017, Gawlowicz et al. [46] presented UniFlex, a wireless network control framework that is able to abstract various protocols and hardware adapters. UniFlex is geared toward node control and management across multiple nodes using a central broker-based abstraction layer. Our work is different architecturally, as it implements more explicit and tighter integration with GNU Radio flowgraph control. Snout is also different in purpose, as it is more geared toward local control of SDR equipment, rather than network management level coordination between multiple nodes across a network.

Batista et al. [47] presented a middleware environment for IoT, which ties together multiple wireless protocols, applications, and cloud services. While their work focuses on communication between individual wireless gateways and the cloud, our work is complementary as it deals with the complexity of software-based implementation of individual wireless protocols as well as their abstraction and encapsulation (i.e., at the gateway level rather than the fog/cloud level). While the architecture proposed in [47] aims at improved interoperability of IoT devices during the development phase of IoT devices by providing common APIs and standardized gateway layers, Snout focuses on reducing the complexity and hardware required to dynamically access arbitrary IoT protocols via SDR tools in an operational setting of existing devices.

González-Barbone et al. [48], [49] proposed a method for adding introspection and control to GNU Radio flowgraphs by extending generic GNU Radio blocks via inheritance in Python, thereby integrating finite state machines, as well as event-based communication between blocks. This method is limited to adding capabilities inside an existing flowgraph, whereas Snout goes far beyond the flowgraph layer and only integrates GNU Radio flowgraphs as components, without requiring customized blocks inside the flowgraph files.

In 2020, Goldsmith et al. [50] presented a demonstrator for a novel way of visualizing and controlling Radio Frequency System on Chip (RFSoC) devices. This work focuses largely on FPGA development inside the RFSoC architecture and the Python-based PYNQ framework used to interact with it, as well as a JupyterLab-based visualization. As such, our work has low direct overlap and could potentially have complementary character, as it focuses on the creation of a general purpose, protocol-agnostic abstraction layer between existing low-layer SDR frameworks and application layer software.

In 2021, Zubow et al. [51] introduced the GrGym middleware that instruments GNU Radio flowgraphs in order to make their internal state accessible for machine learning problems. GrGym demonstrates that instrumentation of GNU Radio flowgraphs using ZeroMQ can be achieved efficiently, performing well even in distributed settings with fairly slow network connectivity. GrGym differs from Snout in that it instrumentalizes GNU Radio flowgraphs by adding specific observation and machine learning-specific blocks to the flowgraph, whereas Snout integrates modularized flowgraph fragments into its higher-level codebase by using hierarchical blocks (see Section 6). Furthermore, GrGym is a middleware with a focus on OpenAI Gym instrumentation of GNU Radio flowgraphs, whereas Snout's architecture allows for generic integration with various existing frameworks via its API.

A preliminary version of this work was presented as a poster in [52], which demonstrated scanning IoT protocols such as Bluetooth LE and Zigbee via a common command-line application. The Snout middleware builds on this work by a fundamentally different, and more flexible ZeroMQ-based approach to data flow handling between GNU Radio components and user-facing application layers, and a clear software architecture definition with decoupled interfaces between the hardware support layers and client applications. Furthermore, this work emphasizes modularity and supports multi-protocol configurations or even multiple SDR frameworks in the same runtime (e.g., mixing GNU Radio component with other signal processing software, see Section V-D).

IV. Software Architecture

In this section, we describe the software architecture of the Snout frameworks, from a module, data flow, and configuration perspective.

A. Module architecture

The Snout software architecture is layered as a middleware, which serves one or more clients via a standardized API, abstracting away the complexity of underlying processes, such as GNU Radio flowgraphs, custom SDR tools, or other arbitrary processes. Figure 3 visualizes the module hierarchy of the software architecture, building up from different hardware components and their respective drivers to different layers of the proposed middleware. The API module defines both the communication protocol (see Section IV-C) as well as a set of base classes that all other components of the software inherit from in a generic way that enables plugin-based integration of third-party variants, which extend its functionality (see Section IV-E). Execution of individual commands, such as running, stopping, or changing parameters of a GNU Radio flowgraph, is managed in a Runtime module, which can be configured using a text-based description language (see Section IV-D).

Inside the Runtime module, task execution is handled by one or more Instruments, which are standardized wrappers

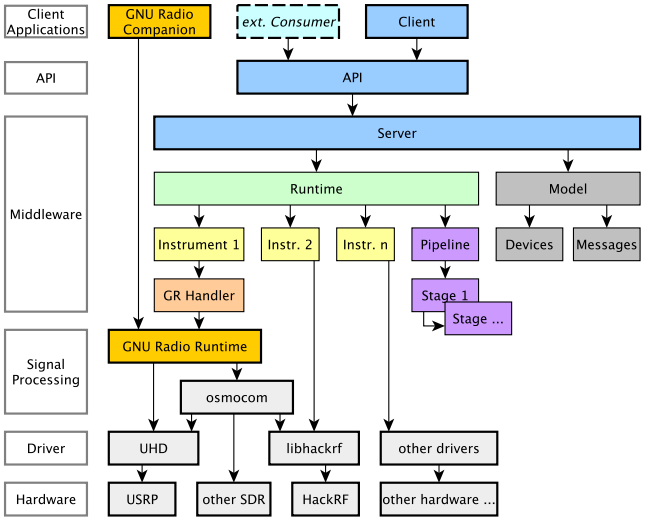

Fig. 3. The Snout module architecture showing the module hierarchy of different components in the system. Note, that arrows indicate composition rather than communication paths. Lower layers support the GNU Radio runtime (pictured in orange) or any other radio protocol implementations supported by the host operating system.

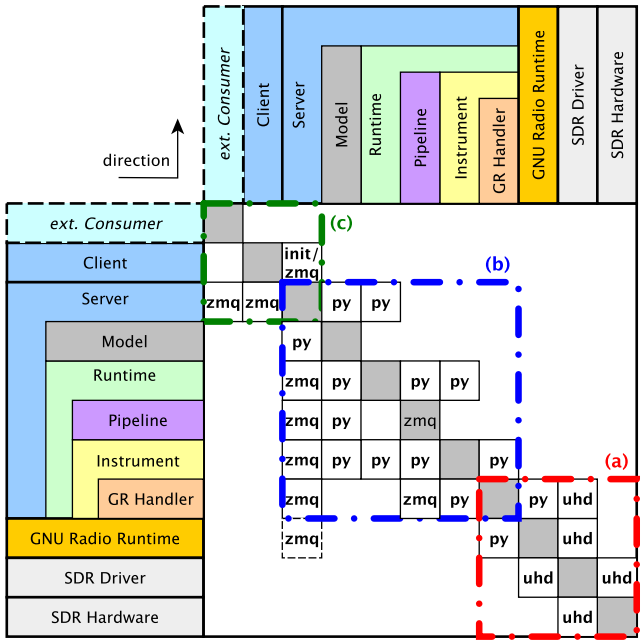

Fig. 4. A Design Structure Matrix (DSM) indicating the ways different architecture components of Snout communicate with each other. Subsystem (a) represents the hardware abstraction layer implemented for GNU Radio [2], (b) represents the Snout middleware environment, and (c) represents the client application layer.

around any kind of sub-process. In their most basic form, Instruments can run arbitrary commands as a shell command and interact with them via `stdin`, as well as collect their `stdout` and `stderr` streams. More elaborate Instrument variants may have specialized ways of interacting with the API of specific applications, e.g., GNU Radio, or protocol-specific software/hardware stacks like srsRAN for LTE / 5G [53].

GNU Radio flowgraphs compile to an executable Python script that implements a so-called "Top Block", which is a data structure in which all elements of the flowgraph are stored and controlled. As Snout is Python-based, implementing direct control over GNU Radio flowgraphs can be achieved without the need of the aforementioned control via shell commands. Instead, the GNU Radio Handler class instantiates a GNU Radio `top_block` class, as well as the GNU Radio blocks for interacting with the SDR hardware (e.g., the `uhd_rx` and `uhd_tx` blocks from the `gr-uhd` [39] module for USRP devices). Then, it imports the desired SDR functionality as a hierarchical block and dynamically assembles a GNU Radio flowgraph that can be controlled from the parent GNU Radio Handler (and subsequently, its parent Instrument class). This is further illustrated in Section V-A. While the GNU Radio Handler is of central importance to Snout as a middleware for controlling GNU Radio flowgraphs, it should be noted that other SDR subsystems, such as the other software radio frameworks mentioned in Section II-B, could be supported by developing a suitable handler class, and by changing the hardware abstraction subsystem (see Section IV-B).

Data received by Instruments from their child processes can be passed into a Pipeline module (see Section IV-C). This allows for heavy processing to occur outside of the GNU Radio flowgraph, which typically is tightly real-time constrained. The Pipeline consists of a series of processing Stages, each of which run their own worker process. Each worker process receives data to work on, performs its task and passes the processed data on to the next Stage.

Finally, a Model class allows for the collection of Message and Device information in a way that allows for Instruments or Pipeline Stages to use traffic analysis statistics (such as device uptime, number of message received per device, etc.) as input to logic gates or processing algorithms.

B. Modular decoupling

A Design Structure Matrix (DSM) is a modeling tool "used to represent the elements comprising a system and their interaction, thereby highlighting the system's architecture" [54]. It can be interpreted as an adjacency matrix of system components, in which matrix elements represent (directed) interactions or dependencies between them. Figure 4 shows a DSM of the Snout framework's main components[3]. Decomposing Snout in this way helps to visualize its decomposition into multiple subsystems.

a) GNU Radio & hardware abstraction: The red box labeled (a) in Figure 4 represents the low-layer interactions between the GNU Radio runtime, the SDR drivers, and the SDR hardware. Notably, Snout only communicates with the GNU Radio runtime using a *GNU Radio Handler* class, which natively controls the GNU Radio flowgraph classes. Furthermore, all hardware-related interaction is encapsulated inside of this subsystem, and is performed by the interaction of GNU Radio [16] and the UHD library [39] or any other

[3]Figure 4 uses the *inputs in rows* (IR) notation used by Eppinger et al. [54], in which an element on the intersection of row i and column j represents an input to component j coming from component i.

supported hardware support layer (e.g., osmocom GNU Radio blocks [55]). If a different hardware platform was desired, changes to this subsystem would not spill over into other parts of the Snout framework, except for the handler class, which directly interacts with it.

b) Snout server: The blue box labeled (b) represents the Snout server subsystem. It exhibits a fairly hierarchical structure, in which all the DSM elements above the diagonal are controlling dependencies, e.g., where the Snout Server class exerts control over the Runtime and Instrument class, and receives status information from the Runtime class in return. An exception to this hierarchy is the Model component, which is queried by Runtime elements, but not directly controlled. All Runtime elements report information to the Snout server via ZeroMQ (denoted `zmq`) communication, which is further described in Section IV-C. An (optional) exception to the decoupling of this subsystem with the underlying GNU Radio and hardware subsystem is the fact that the GNU Radio runtime is capable of sending information from the flowgraph via ZeroMQ by using the `gr-zeromq` module [16], which may be useful for low-layer data, e.g., flowgraph debugging messages.

c) Snout client(s): The green box labeled (c) represents the Snout client as well as any external application, which receives input from Snout (i.e., *consumer*). In order to decouple the external interface of Snout with hardware, GNU Radio and Snout's internal structure, the Snout API provides information to these programs in a standardized way while allowing the client to control the server (see Section IV-C).

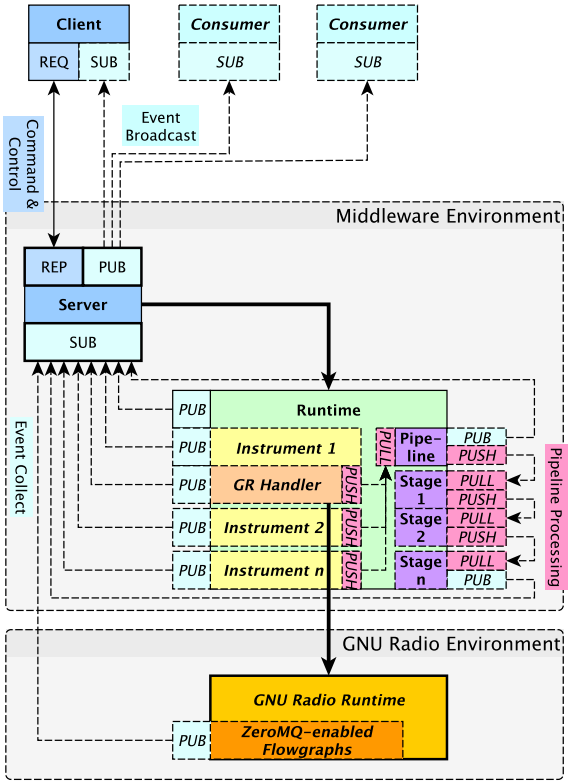

Fig. 5. ZeroMQ-based data flow between the Snout Server and Client. Dashed lines represent optional components. Wide black arrows represent data flows and communication via native Python-based interaction without using ZeroMQ.

C. Data flow architecture

The data flow between different modules of Snout is implemented using ZeroMQ [56], which is a brokerless messaging library for distributed systems.

While Snout data flow could be implemented by a number of available messaging systems, ZeroMQ was chosen for three key reasons. First, ZeroMQ has stable support for GNU Radio via the official `gr-zeromq` package that is distributed with the GNU Radio distribution [16]. Further, as Kang et al. note [57], ZeroMQ performs extremely well in terms of throughput and latency for small message payloads[4] when compared to other popular message-passing systems like the OMG Data Distribution Service (DDS) or MQ Telemetry Transport (MQTT). Last, as a brokerless system, ZeroMQ does not suffer from the additional dependency, overhead, and potential bottleneck of a message broker [57].

ZeroMQ enables a number of flexible message queueing patterns that can be used within a process, across processes on the same machine, or between applications running on remote hosts. It allows applications to communicate in a thread-safe and language-agnostic way. Two or more nodes can connect to a common ZeroMQ socket using multiple communication patterns. Similar to plain socket-based communication, one node *binds* to a socket, and one or more additional nodes can dynamically *connect* to the same socket in order to communicate. The key patterns used in the context of this work are (see also Hintjens, Ch. 2 [58]):

- **Request-Reply.** This a typical client-server configuration in which a client makes a request to one or more servers, and receives a reply. If multiple servers are available, the client will make each request to one of the servers in a round-robin fashion.
- **Push-Pull.** This pattern allows nodes to push messages to one or more downstream nodes, and receive messages from one or more upstream nodes.
- **Publish-Subscribe.** In this pattern, one or more nodes publish messages that other nodes can subscribe to. This pattern typically utilizes multi-part messages, in which the first part of the message is a subscription channel identifier (such as `logging.DEBUG` to indicate a debugging message). Subscribers can choose to receive all messages distributed by the publisher, or just those matching a certain channel value.

More advanced patterns are possible, as described in the ZeroMQ documentation [56] and in the official guide by Hintjens [58].

Communication between Snout components can be broken down into three major categories:

[4]For message sizes below 1 kilobyte, Kang et al. find that ZeroMQ performs two orders of magnitude faster than OMG DDS and three orders of magnitude faster than MQTT [57]. As typical IoT wireless packets are usually on the order of a hundred kilobytes or smaller in size, this makes ZeroMQ well-suited for a middleware that handles small control messages as well as typical IoT packet messages.

- **Command & Control** data flows, between the *Client* and the underlying *Server*. This comprises a number of explicit control commands (such as `init`, `run`, `stop`, etc.) or requests for information on the current system status (e.g., `status`).
- **Event** data flows, which can be produced by any of the modules inside of the Snout server runtime, and may be consumed by the Client or additional *Consumers*.
- **Pipeline** messages, which are handed from one processing Stage of the Pipeline to the next in a Push-Pull pattern (see Figure 5). The same pattern is used to push messages from Instruments to the start of the Pipeline.

The case of command & control data flow is a typical client-server scenario in which the client makes a *request* to a server, which handles the request and returns a *response*. This is implemented in a Request-Reply pattern. The server binds to a socket in `REP` (Reply) mode and remains available for its entire duration of service. A client then connects to this socket in `REQ` (Request) mode to issue commands to the server.

Event data requires a more complex messaging architecture, as many of the components that may produce events, as well as any event consumers are optional (indicated by dashed lines in Figure 5), and the overall system must function reliably with any of them disconnecting. To ensure reliable distribution of events in the context of potentially transient event producers and consumers, the server component always binds to a socket and allows any peripheral component to connect or disconnect at any time. Using a "Pub-Sub Network with a Proxy" pattern [58], the server binds to a socket that we refer to as *event collector socket* in `SUB` (Subscriber) mode, and binds to an *event broadcast socket* in `PUB` mode. Then, any number of event producing components, such as the Runtime, any number of Instrument instances, or even underlying GNU Radio components, can connect to the event collector socket and publish arbitrary event messages. The server then forwards all received events on its own event broadcast socket, reliably forwarding them to the client or any other event consumer (such as logging or database applications).

Event messages are multi-part messages, in which the first part always represents a channel key, and the second part is the actual event payload. The channel identifies the type of data and dictates the data type that is used in the event payload, e.g., in case of a logging event, the channel may be "`log.INFO`", and the event payload is simply the log message as a string. In more complex cases, the channel could indicate a data frame received via GNU radio, in which case the channel may be "`rx.ble`" and the event payload could be the decoded byte stream of the frame. These channel identifiers allow user applications to subscribe to events of a certain type, e.g., a message transcript window may subscribe to decoded packets, a status terminal may subscribe to logging information and runtime control message, and so forth. Crucially, these applications are not required to be known to, or even explicitly compatible with Snout. Instead, messages broadcast in this way are interoperable with any software that supports subscribing to ZeroMQ publishers.

As shown in Figure 5, Pipeline stages are connected by a series of Push-Pull pattern message queues. This allows for individual Stages to be executed either locally or on a remote processing host, and it supports fan-out patterns to multiple workers that can process a stage if certain workloads would otherwise be a performance bottleneck.

```yaml
---
meta:
  name: Bluetooth LE Physical Layer Receive
  description: >
    This is a basic Bluetooth LE scanning setup.
  author: A. N. Onymous
  email: name@example.com
  date: 2021-05-21

instrument:
  bleadv:
    class: gnuradio
    handler:
      - bleadv
    output: pipeline

parameters:
  channels:
    - 37
    - 38
    - 39
  timeout: 10

runs: 3

steps:
  - run:
      instrument: bleadv
      command: run
  - stop:
      instrument: bleadv
      command: stop
      condition:
        type: time elapsed
        criteria: timeout

pipeline:
  - bleadv_linklayer
  - blepdu_scapy
```

Listing 1. Example of a Snoutfile illustrating the standard schema used to describe Snout configurations.

D. Description language

Snout configurations can be developed using a standardized format, which we refer to as a *Snoutfile*. Snoutfiles are based on a subset of the YAML 1.2 [41] data serialization language, which contain all necessary definitions of the composition of the runtime, as well as definitions of a sequence of steps that will be executed at runtime. Listing 1 shows an example of a Snoutfile.

A Snoutfile follows a schema of top level keywords containing configuration information for each component of the Snout framework. The schema is enforced by using the StrictYAML [59] parser, which ensures type safety of the configuration and mitigates security concerns commonly associated with the default implementation of YAML in Python [60]. The schema defines the following sections:

- **meta:** Metadata about the configuration, such as title, description, and author information.
- **instrument:** Contains a dictionary of named instruments. The output of each instrument can be configured (in

```
[options.entry_points]
smp_plugins =
  instrument_ = smp.instr:Instrument
  instrument_process = smp.instr:ProcInstrument
  instrument_gnuradio = smp.instr:GRInstrument
```

Listing 2. An excerpt of entry point definitions for Instrument variants.

this example it points at the Pipeline for further post-processing of the incoming data). In Listing 1, the dictionary contains one instrument named `bleadv` which is a `gnuradio` Instrument using ad GNU Radio Handler with the identifier `bleadv`.

- **parameters:** Defines parameters that can be passed to the runtime, and used by instruments. Parameters with values indicated as lists can be used to iterate through multiple values (such as `channels` 37, 38, and 39 in Listing 1). In this case, the instrument will be run with three sequential parameter sets `(37, 10)`, `(38, 10)`, `(39, 10)` to cover all desired channels for a duration of 10 seconds each.
- **runs:** Indicates the number of times the individual executions resulting from the given parameters should be repeated. This is useful if Snout is used to run a repeatable experiment, and multiple runs of the exact same sequence of actions are desired for statistical analysis.
- **steps:** Contains a list of steps to be executed by an instrument when certain conditions are met. In Listing 1, two steps are defined, the first of which executes the `run()` method on the `bleadv` Instrument at time $t = 0$, and the second executes the `stop()` method on the `bleadv` Instrument at $t =$ `timeout` (referring to the variable defined in the `parameters` Section of the file).
- **pipeline:** defines a list of Pipeline Stages. In Listing 1, the received byte-stream of Bluetooth LE advertising frames is sent from the `bleadv` instrument to the beginning of the pipeline, where the `bleadv_linklayer` Stage performs the necessary de-whitening of the payload, and subsequently the `blepdu_scapy` parses the byte-stream to a scapy [61] packet, which is then broadcast via the event collection mechanism described in Section IV-C.

E. Plugin-based extensions

Snout uses Python's entry points mechanism [62] to define variants for classes such as Instruments or Pipeline Stages. This allows new components to be developed in a separate project, and the correct class variation to be imported at runtime via a Factory pattern [63].

Listing 2 shows an abbreviated list of entry point definitions for variants of Instrument classes. When the Runtime is constructed from the Snoutfile, the `class` identifier of an instrument (e.g., `gnuradio` in Listing 1, Line 12) is concatenated with the generic class name (`instrument`) to look for a suitable specialized class. In the case of Listing 2, looking up `instrument_gnuradio` will return a reference to the appropriate `GRInstrument` class which is defined in the `snout.instr` module.

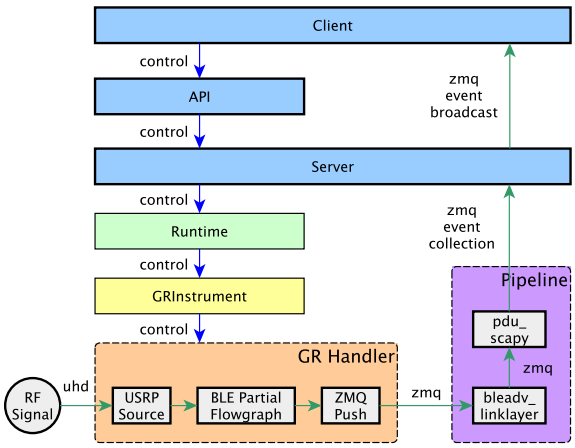

Fig. 6. A Snout-based application scanning BLE traffic. Blue lines indicate control flow, green lines indicate data flow of the received traffic.

Protocol	Device ID	Vendor	Last Seen	#	Uptime
ble	5f:c7:92:	Apple, Inc.	18 seconds ago	16	2 minutes
ble	69:12:80:	-	a minute ago	3	instantly
ble	65:6d:ce:	-	a minute ago	2	24 seconds
ble	1e:8b:03:	-	19 seconds ago	14	2 minutes
ble	4c:dc:f7:	Apple, Inc.	20 seconds ago	29	2 minutes
ble	f6:40:6c:	Fitbit Inc.	a minute ago	3	21 seconds

Fig. 7. A simple command-line application collecting device statistics based on incoming scan traffic collected using a Snout-based BLE scanning application.

V. Validation and Evaluation

In this section, we present four representative case studies to validate key capabilities supported by Snout. We demonstrate how one or multiple protocol implementations, and even multiple distinct software stacks can be integrated with Snout, and evaluate its performance overhead.

A. Implementation of a BLE advertising scanner

We first demonstrate a BLE advertising scanner application, as shown in Listing 1. In order to integrate SDR-based BLE advertising into the Snout framework, only small modifications to the flowgraph shown in Figure 2 are required: the *USRP Source*, as well as the *ZMQ PUSH Sink* blocks are removed, and replaced with the *Pad Source* and *Pad Sink* blocks[5], respectively. This partial flowgraph is compiled to a *hierarchical block* (as opposed to a typical *top block* that can be run directly from the GNU Radio Companion or as a standalone application). The resulting block can then be imported to the GNU Radio Handler component. The GNU Radio Handler maintains its own generic top block and USRP Source block, and connects the output of the source block to the imported BLE receiver partial flowgraph, and its output to a ZeroMQ PUSH block, which relays its output to the processing pipeline.

[5]Pad Source and Pad Sink blocks are GNU Radio blocks that can either be attached to the beginning (source) or end (sink) of a flowgraph to terminate it. A flowgraph that contains pad sources or sinks can be used as a subcomponent of a larger flowgraph, with its pad source blocks acting as block inputs, and pad sinks acting as block output. Snout leverages this GNU Radio functionality to compose a top level flowgraph at runtime that individual flowgraphs with pads are plugged into.

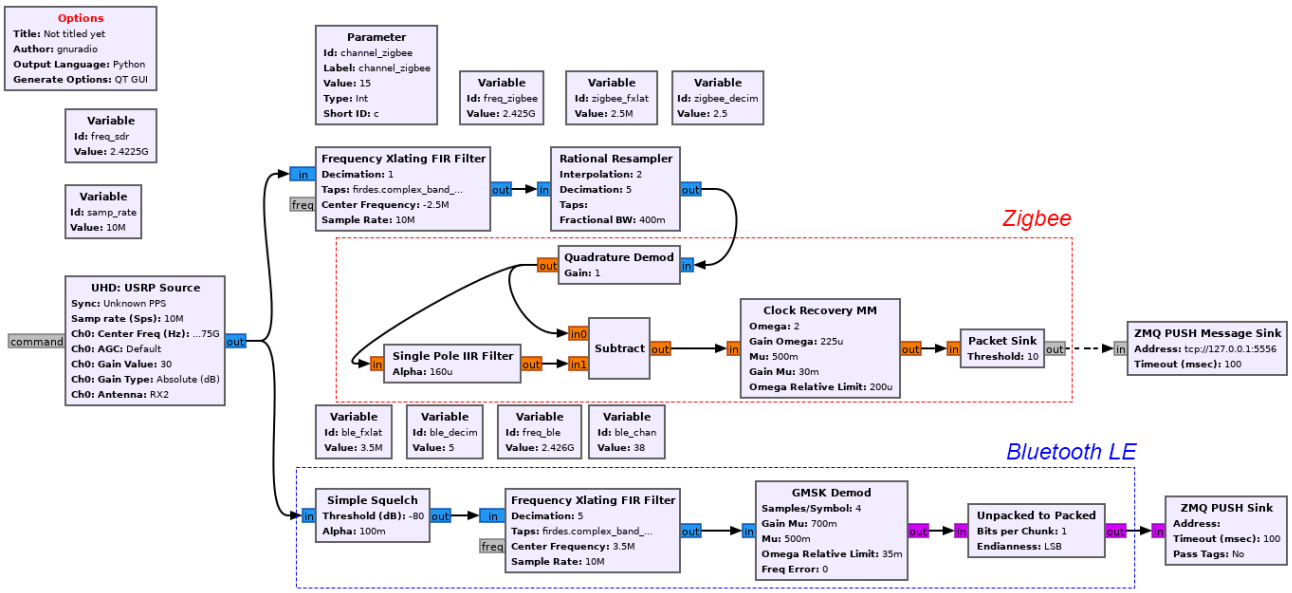

Fig. 8. A GNU Radio flowgraph implementing two parallel receivers for different protocols, Zigbee (red) and BLE (blue). The complexity of this flowgraph lies in the signal processing as such, as well as the numerous variable blocks, which may contain additional (hidden) calculations. Snout aims at reducing the complexity and visual clutter of such a flowgraph to the two protocol implementations shown in dashed boxes, and configures everything around it automatically at runtime.

Functionally, this flowgraph is now performing the exact same task as as the flowgraph in Figure 2, namely receiving and demodulating a physical layer signal to byte-streams of BLE advertising frames. However, since the USRP Source as well as the output blocks are controlled by the middleware, any of its settings can be configured in the Snoutfile, and further adapted by custom instrument or handler classes, which do not have to touch the GNU Radio flowgraph fragment. Figure 6 shows the control flow (blue) and data flow (green) of the resulting application. Figure 7 shows the output of an independent command-line client monitoring BLE devices based on this Snout-based implementation.

B. Implementation of multi-protocol scanning

Figure 8 illustrates the complexity of a pure GNU Radio-based minimal implementation of BLE and Zigbee parallel receiver in the GNU Radio Companion, which is a typical use case for multi-protocol IoT reconnaissance (e.g., *IoT-Scan* by Gvozdenovic et al. [64]). In such a scenario, multiple protocol-specific flowgraph components are combined to receive traffic from several channels and/or protocols at the same time. In addition to the visual complexity of the flowgraph in Figure 8 based on the connectivity between its processing blocks, many of the variable blocks (i.e., the blocks that do not have any edges going in or out of them) can hold either static values, or calculation formulae, which depend on an arbitrary number of other blocks. Constructing such a flowgraph requires in-depth signal processing skills. Furthermore, managing the evolution of such a block is complex due to the hidden logic and values behind each block.

When interacting with GNU Radio flowgraphs directly, any configuration change usually has to be implemented by manually adding, moving, and connecting new blocks to a flowgraph

```
instrument:
  multiscan:
    class: gnuradio
    handler:
      - bleadv
      - zigbee
    output: pipeline
```

Listing 3. Snoutfile changes implementing a multi-protocol scanning instrument comprising of a BLE advertising scanner as well as a Zigbee scanner.

to achieve the desired functionality. The partial flowgraph abstraction discussed previously in Section V-A allows for the creation of single-purpose flowgraph fragments with a standardized interface (e.g., based on the sub-flowgraphs noted "Zigbee" and "Bluetooth LE" in Figure 8). These partial flowgraphs can be designed individually and without the need for managing the auxiliary elements required to piece them together into a combined flowgraph. Snout builds the additional elements at runtime based on information provided in the Snoutfile.

Using Snout, adding another wireless protocol, e.g. Zigbee, to the previous application therefore only requires two modifications: (a) the creation of a partial flowgraph, e.g., based on the IEEE 802.15.4 transceiver by Bloessl et al. [65] with pad source and sink blocks, and (b) changing the Snoutfile to add any desired configuration parameters (e.g., as Zigbee channels) as well as the additional GNU Radio Handler component to the configuration. Listing 3 shows the modified instrument definition accommodating two GNU Radio Handlers and Figure 9 shows the resulting data flow.

This not only reduces the visual clutter of complex GNU Radio Companion projects, but furthermore allows these simple building blocks to be simultaneously and automatically

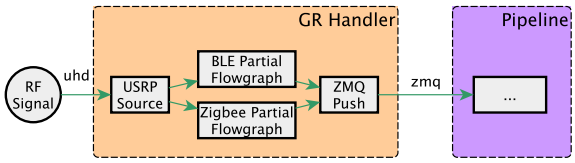

Fig. 9. A Snout-based application integrating both BLE and Zigbee scanning in a GR Handler block, connecting both protocols to the USPR Source dynamically at runtime.

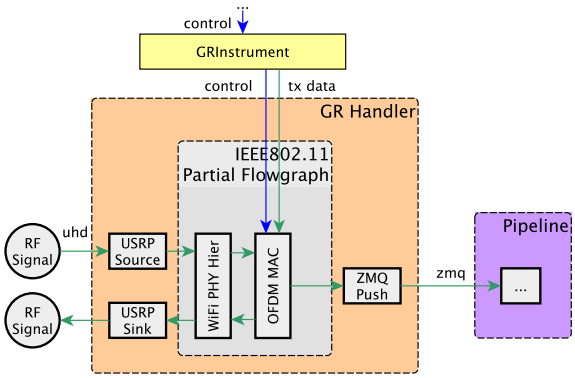

Fig. 10. A Snout-based application integrating the IEEE 802.11 transceiver library by Bloessl et al. [42], shown inside the grey box.

controlled by the runtime logic (whereas GNU Radio Companion only allows the user to manually start or stop a flowgraph). This is crucial in experimental setups that require repeatable runs or complex parameter variation, e.g., permutations of multiple gain and delay values across a large number of parametrized runs, as required for experimental setups [10].

C. Encapsulating a Wi-Fi testbed

Bloessl et al. [42] presented a GNU Radio-based IEEE 802.11 transceiver testbed, which allows for SDR-based experimentation on Wi-Fi networks, including the 802.11p variant for vehicular ad-hoc networks. The reference flowgraph provided by Bloessl et al. primarily consists of a hierarchical block reponsible for the physical layer, as well as an OFDM MAC layer block (see Figure 10). Snout can encapsulate this library to facilitate SDR-based Wi-Fi monitoring without requiring the user to be familiar with the development of software radio flowgraphs. In contrast to the previous case study, Snout provides a standardized USRP Sink block in addition to the USRP Source block, both of which interface with the physical layer hierarchical block that is provided by the library [42] and handle transmission and reception of the signal.

D. Integrating different toolchains

Previous case studies presented different variations of GNU Radio-based tools addressing different use cases. As shown in Figure 3, Snout is not limited to GNU Radio-based SDR implementations, and different toolchains can be combined without changing high level behavior.

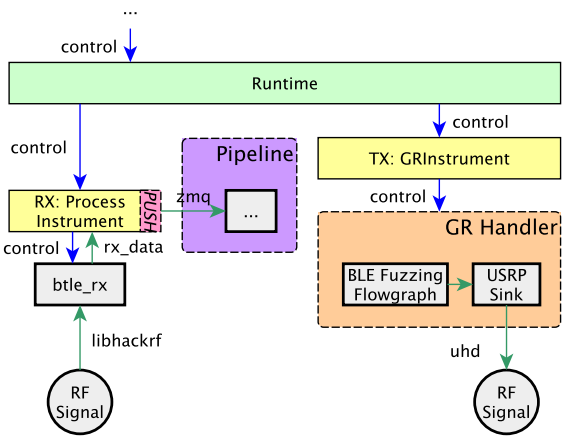

Fig. 11. A heterogeneous configuration in which both a GNU Radio-based instrument, as well as standalone binary controlled by a `ProcessInstrument` are used by a common runtime.

One example of such a use case is an experimental setup for BLE physical layer fuzzing. Fuzzing, i.e. the introduction of arbitrary disturbances and malformed input to a system in order to discover unexpected output, can be applied to wireless protocols to discover undefined protocol behavior, security vulnerabilities, and bugs (we refer to McNally et al. [66] for an overview on fuzzing and to Knight and Speers [8] for an introduction on radio frequency (RF) fuzzing). Fuzzing wireless protocols requires control over the entire protocol stack, including the capability to produce signals that are not protocol-compliant. This can be achieved by using tools such as the GNU Radio-based TumbleRF [8]. In such a case, one would implement the transmission part of the experiment based on GNU Radio and TumbleRF (or other suitable fuzzing tools), whereas one could rely on a simple standalone binary on the receiver side, such as Xianjun Jiao's BTLE [67] as implemented in Becker et al. [68].

Such a setup employs multiple instruments, namely one `GRInstrument` for the transmission (TX) chain, and a separate `ProcessInstrument` controlling the receiver (RX) binary, as shown in Figure 11. This allows for an arbitrarily complex fuzzing toolchain based on GNU Radio flowgraphs controlled via the TX instrument, while a simple and low-resource tool like the `btle_rx` binary [67] is used to process incoming packets. Notably, switching between this receiver toolchain and the GNU Radio-based described in Section V-A (cf. Figure 6) can be performed without technical knowledge of their respective inner workings, by simply modifying the Snout file.

E. Experiments and performance evaluation

1) CPU and memory overhead: GNU Radio flowgraphs are complex applications with considerable resource demands. Executing a flowgraph imposes certain real-time constraints, which, if not respected, result in dropped packets due to buffer over- or underruns within components of the flowgraph. It is therefore important to demonstrate that a middleware

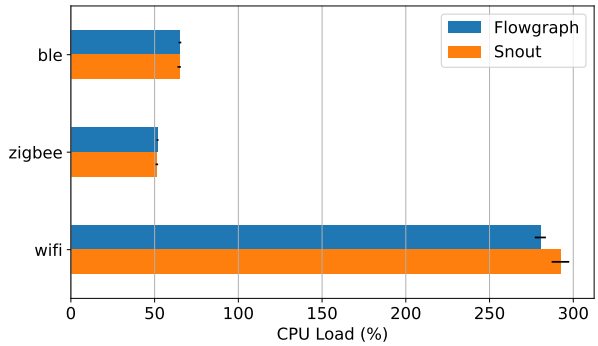

Fig. 12. Running different wireless protocols via Snout compared to a standalone GNU Radio flowgraph does not incur significant overhead in CPU load, as measured on a 6-core system. The results are consistent across experiments, as shown the by tight 95% confidence intervals (black bars).

controlling GNU Radio flowgraphs does not incur significant processing overhead, which may negatively impact its real-time signal processing performance.

To investigate this potential overhead, we measure and compare the resource consumption of Snout running different flowgraphs with the resource consumption of only running the flowgraphs directly via GNU Radio[6]. We perform the measurements by running each setup inside a Docker container, and measuring CPU and RAM usage externally via the `docker stats`[7] command, which measures per-core loads of a container (i.e. up to 100% per physical core). All experiments are performed on a PC equipped with an AMD Ryzen 5 3600 CPU with 6 physical cores (12 threads), with 32GB DDR4 RAM, PCIe4 NVMe storage, and running 64-bit Linux kernel 5.10. The Docker container is based on the official Ubuntu 20.04 image and the official GNU Radio 3.8 installation packages [69] as well as the `gr-ieee802-15-4` [65] and `gr-ieee802-11` [42] packages.

As shown in Figure 12, Snout does not incur measurable CPU overhead when running low-bandwidth IoT protocols like Zigbee and Bluetooth LE, compared to the bare GNU Radio flowgraph. Each of the measurements is based on 600 samples (10 runs of 60 seconds) with very tight 90% confidence intervals, which shows that both the pure GNU Radio flowgraph and the Snout-controlled flowgraph produce a consistent, sustained processing load during execution. For Wi-Fi, measurements using the Wi-Fi transceiver of Bloessl et al. [42] indicate a small but measurable performance overhead of about 4%, which may be attributed to the size of 802.11 frames. This is an expected result, as ZeroMQ performs best with small packets [57]. We view this overhead as unproblematic given the overall load below 300% on a 6-core system (i.e., only utilizing half of the physical cores, and a quarter of the available threads).

Snout does incur an increased memory requirement, necessitated by the data structures managing runtime control as well as message passing. In the above test cases, memory overhead was consistently in the range of 80 to 90MB above the memory footprint of the GNU Radio flowgraph alone, which ranged from around 73MB for the BLE and Zigbee flowgraphs and around 112MB for the Wi-Fi flowgraph. On a current system with available memory in the order of multiple Gigabytes, this does not raise serious concerns.

[6]The GNU Radio flowgraph is run in command-line mode in order to prevent any GUI overhead impacting the experiment.

[7]See https://docs.docker.com/engine/reference/commandline/stats/.

Fig. 13. The command and control message system (a subset of the data flow laid out in Figure 5).

2) Message handling performance: Aside from overall system load overhead incurred by Snout measured in the previous Section, a performance evaluation of its internal message handling system can provide additional insights into middleware performance in terms of message throughput and latency.

a) Command & Control message handling: As previously introduced, Snout uses a request-reply pattern for its command and control handling. Investigating this message passing subsystem in isolation (see Figure 13), we can measure the performance impact of passing commands into the Snout middleware. While Kang et al. [57] demonstrated that ZeroMQ excels at small message sizes, is it important to investigate behavior at various message sizes to prevent performance issues in case of larger command messages.

We consider message sizes of $10^1, 10^2, \ldots, 10^5$ bytes. We send 100 command messages for each size in the fastest possible succession (i.e., as soon as acknowledgement of a command is received, the next command is sent). This experiment is repeated 10^3 times in order to stress-test the system and produce a significant amount of data points. For each message size, we measure the mean and the 95% and 99% quantiles of the round-trip latency to quantify message-passing performance. The results are shown in Figure 14. For messages of small size, a single command only requires about $65\,\mu s$ roundtrip time (i.e., from sending the message to receiving an acknowledgement of the server) with the 99% quantile remaining below $100\,\mu s$. These statistics remain fairly constant up to the 10^4 byte message length mark, and then increase visibly. However, despite such a significant increase in message size, roundtrip latency remains at a mean of around $110\,\mu s$, and a 99% quantile of just over $140\,\mu s$.

In a typical experiment, these command and control messages correspond to steps defined in the Snoutfile or status requests, both of which would typically not be performed with large amounts of messages. It can therefore safely be assumed that the command and control message subsystem contributes a negligible amount of load to the overall system.

b) Event message handling: The message passing subsystem transporting events from their respective source in the Snout runtime via the server to data consumers may potentially produce a large number of messages based on received traffic. We consider a system with five event producers and three event subscribers, as shown in Figure 15.

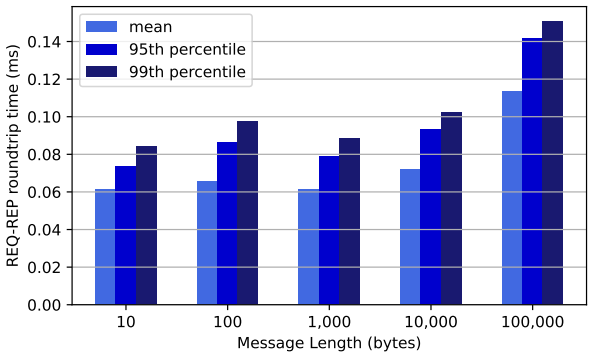

Fig. 14. Roundtrip times between client and server remain consistently low for message lengths of up to $10,000$ bytes.

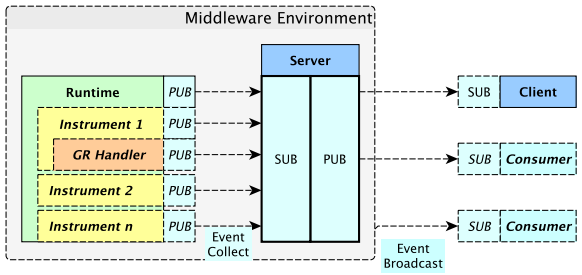

Fig. 15. The event message message system (a subset of the data flow laid out in Figure 5).

Again, we consider sizes of $10^1, 10^2 \ldots, 10^5$ bytes per event message. Each event producer embeds the creation timestamp of an event into the message transferred to the event subscribers, which calculate message latency based on their timestamp of receipt. Each producer generates 10^4 event messages at a rate of 10^3 messages per second (i.e., a total of 50×10^3 event messages) for each message size and publishes them.

Figure 16 shows the results of overall event message latency, again measuring the mean, 95%, and 99% quantiles. For events of sizes up to 10^3 bytes, the mean latency stays at around $200\,\mu s$ with a 99% quantile around $320\,\mu s$. Event messages

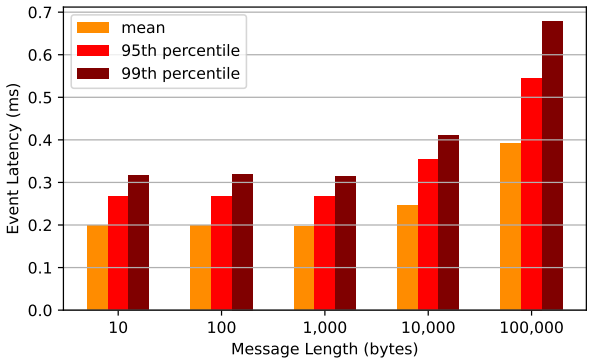

Fig. 16. Total latency of event messages arriving at an event consumer is $0.2\,\mu s$ on average for event messages up to 1,000 bytes in length, and roughly doubles for event messages of 100,000 bytes in length.

of 10^4 bytes length experience a performance deterioration of about 25% and messages of 10^5 bytes take about twice as long as the shortest messages. All in all, this demonstrates throughput of 5×10^3 event messages per second relayed to their respective consumers in well under a microsecond, i.e., an order of magnitude faster than graphical user interfaces on high refresh-rate $120\,Hz$ monitors would be able to notice. It should be noted that such a large number of messages – synthetically generated – are a very brute-force stress test that would rarely make sense to perform in practice. For reference, the resulting $500\,Mb/s$ corresponds to three times the raw sample output of a HackRF One [5] (20 mega samples per second times 8 bits per sample) or two thirds of the entirety of raw sample data of a USRP B200 [70] (61.44 mega samples per second times 12 bits per sample) streamed to three consumers. Considering the typical GNU Radio setup in which raw sample data is processed within the flowgraph and only aggregated information (such as demodulated binary signals) is passed on to downstream applications, this guarantees almost imperceptible and consistently low event message latency in virtually all imaginable scenarios.

VI. Conclusion

In this work, we presented Snout, a middleware that encapsulates SDR-based protocol implementations and enables applications based on GNU Radio flowgraphs that do not require digital signal processing skills from the user. Its modular software architecture allows for flexibility and expansion on all three layers: First, client applications can connect to the middleware in a standardized way, and subscribe to relevant data via a ZeroMQ subscription pattern. Second, a text-based description language facilitates changes in both instrument as well as pipeline behavior without requiring any code modifications, while entry point-based plugin loading of class variants enables third party code to extend functionality. Last, the hardware abstraction layer supports a number of SDR devices, and can be extended by the GNU Radio community without impacting the top layers of the framework. In addition, it can operate with building blocks made of GNU Radio flowgraphs, or any other SDR application via the use of customizable `Instrument` abstractions. We validated the architecture and evaluated its performance with several popular IoT protocols, demonstrating that it has no negative effect on the computational load of the host computer, limited memory overhead, and that the message passing system can sustain message volumes of typical real-world applications with sub-microsecond latency.

We envision that Snout will constitute a platform supporting experimental work in wireless research, which will facilitate security auditing, wireless device benchmarking, and physical layer characterization. Additionally, graphical user interface concepts for high level monitoring and control of SDR-based tasks could be explored, such as web-based interfaces proposed by Goldsmith et al. [50], which allow for local or remote operation of the SDR instrumentation. Furthermore, the current GNU Radio-based hardware abstraction layer could be extended to support other popular SDR frameworks, such as srsRAN [53].

REFERENCES

[1] E. Blossom, "GNU Radio: Tools for Exploring the Radio Frequency Spectrum," *Linux J.*, vol. 2004, no. 122, p. 4, Jun. 2004.

[2] The GNU Radio Project, "GNU Radio. The Free & Open Software Radio Ecosystem." [Online]. Available: https://www.gnuradio.org/

[3] C. Laufer, *The Hobbyist's Guide to the RTL-SDR: Really Cheap Software Defined Radio*, 3rd ed. CreateSpace Publishing, 2015.

[4] Ettus Research, "Products," 2021. [Online]. Available: https://www.ettus.com/products/

[5] Great Scott Gadgets, "HackRF One," 2014. [Online]. Available: https://greatscottgadgets.com/hackrf/

[6] Lime Microsystems Ltd, "Lime Microsystems - Products - Board Level," 2020. [Online]. Available: https://limemicro.com/products/boards/

[7] Nuand, "BladeRF," 2021. [Online]. Available: https://www.nuand.com/bladerf-2-0-micro/

[8] M. Knight and R. Speers, "Designing RF Fuzzing Tools to Expose PHY Layer Vulnerabilities," in *DEF CON 26*, 2018. [Online]. Available: https://media.defcon.org/DEF%20CON%2026/DEF%20CON%2026%20presentations/DEFCON-26-Matt-Knight-and-Ryan-Speers-Designing-RF-Fuzzing-Tools-to-Expose-PHY-Layer-Vulns-Updated.pdf

[9] FOSDEM '16, "Interview with tom rondeau. gnu radio for exploring signals. talk hard: A technical, historical, political, and cultural look at fm," 1 2016. [Online]. Available: https://archive.fosdem.org/2016/interviews/2016-tom-rondeau/

[10] J. K. Becker, S. Gvozdenovic, L. Xin, and D. Starobinski, "Testing and fingerprinting the physical layer of wireless cards with software-defined radios," *Computer Communications*, vol. 160, pp. 186–196, Jul. 2020.

[11] X. Jiao, W. Liu, M. Mehari, M. Aslam, and I. Moerman, "openwifi: a free and open-source ieee802.11 sdr implementation on soc," in *VTC2020-Spring*, 2020, pp. 1–2.

[12] T. Pereira, M. Violas, J. Lourenço, A. Silva, and C. Ribeiro, "An FPGA Implementation of OFDM Transceiver for LTE Applications," *International Journal on Advances in Systems and Measurements*, vol. 6, no. 1, pp. 224–234, 2013.

[13] J. Bishop, J.-M. Chareau, and F. Bonavitacola, "Implementing 5G NR Features in FPGA," in *2018 European Conference on Networks and Communications (EuCNC)*. IEEE, Jun. 2018, pp. 373–9. [Online]. Available: https://ieeexplore.ieee.org/document/8443214/

[14] The Mathworks Inc., "Communications Toolbox." [Online]. Available: https://www.mathworks.com/products/communications.html

[15] National Instruments, "Software Defined Radio Device Bundle." [Online]. Available: https://www.ni.com/en-us/shop/hardware/products/software-defined-radio-device-bundles.html

[16] The GNU Radio Project, "gnuradio/gnuradio." [Online]. Available: https://github.com/gnuradio/gnuradio

[17] J. Blum, "Pothosware," 2013. [Online]. Available: http://www.pothosware.com/

[18] ——, "SoapySDR," 2015. [Online]. Available: https://github.com/pothosware/SoapySDR/wiki

[19] Redhawk SDR, "REDHAWK," 2016. [Online]. Available: https://redhawksdr.org/

[20] V. A. Sergeev, "LuaRadio," 2016. [Online]. Available: https://luaradio.io/

[21] C. Schmidt, "QIRX SDR," 2017. [Online]. Available: https://qirx.softsyst.com/

[22] C. J. Cliffe, "CubicSDR," 2013. [Online]. Available: https://cubicsdr.com/

[23] G. J. Carracedo Carballal, "SigDigger." [Online]. Available: https://batchdrake.github.io/SigDigger/

[24] J. Ketterl, "OpenWebRX," 2020. [Online]. Available: https://www.openwebrx.de/

[25] A. Csete, "Gqrx SDR," 2013. [Online]. Available: https://gqrx.dk/

[26] K. Reid, "ShinySDR," 2013. [Online]. Available: https://shinysdr.switchb.org/

[27] C. Daniel, Hoernchen, and D. Stolnikov, "Project sdrangelove," 2016. [Online]. Available: https://osmocom.org/projects/sdr/wiki/Sdrangelove

[28] E. Griffiths, "Sdrangel," 2016. [Online]. Available: https://github.com/f4exb/sdrangel/wiki

[29] M. Walters, "inspectrum," 2015. [Online]. Available: https://github.com/miek/inspectrum

[30] B. Reiser, "SodiraSDR." [Online]. Available: http://www.dsp4swls.de/sodirasdr/

[31] L. Åsbrink, "Linrad," 2014. [Online]. Available: http://www.sm5bsz.com/linuxdsp/linrad.htm

[32] SDR Apps, "Studio1," 2011. [Online]. Available: http://www.sdrapplications.it/

[33] A. Di Bene, "HDSDR," 2009. [Online]. Available: http://www.hdsdr.de/

[34] A. Musceac, "QRadioLink," 2017. [Online]. Available: http://qradiolink.org/

[35] 3dB Labs Inc., "SCEPTRE. Realize the full potential of your equipment," 2018. [Online]. Available: http://www.3db-labs.com/software-development

[36] Epiq Solutions, "Skylight: Flexible Wireless Network Survey Application." [Online]. Available: https://epiqsolutions.com/rf-sensing/skylight/

[37] The GNU Radio Project, "GNU Radio Recipes Repository – gr-recipes." [Online]. Available: https://github.com/gnuradio/gr-recipes

[38] ——, "gr-etcetera – Extended set of GNU Radio-related recipes for PyBOMBS." [Online]. Available: https://github.com/gnuradio/gr-etcetera

[39] Ettus Research, "The USRP™ Hardware Driver Repository." [Online]. Available: https://github.com/EttusResearch/uhd

[40] Drtyhlpr, "ble_dump - SDR Bluetooth LE dumper," 2016. [Online]. Available: https://github.com/drtyhlpr/ble_dump

[41] YAML Language Development Team, "YAML: YAML Ain't Markup Language™," 2009. [Online]. Available: https://yaml.org/

[42] B. Bloessl, M. Segata, C. Sommer, and F. Dressler, "Performance Assessment of IEEE 802.11p with an Open Source SDR-based Prototype," *IEEE Trans. Mobile Comput.*, vol. 17, no. 5, pp. 1162–1175, May 2018.

[43] M. Braun, Devnulling, and W. Gebers, "ControlPort," 2018. [Online]. Available: https://wiki.gnuradio.org/index.php/ControlPort

[44] Chunxiao Li, A. Raghunathan, and N. Jha, "An architecture for secure software defined radio," in *Des. Autom. Test Eur.* IEEE, Apr. 2009, pp. 448–453. [Online]. Available: http://ieeexplore.ieee.org/document/5090707/

[45] P. Shome, Muxi Yan, S. M. Najafabad, N. Mastronarde, and A. Sprintson, "CrossFlow: A cross-layer architecture for SDR using SDN principles," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE, Nov. 2015, pp. 37–39. [Online]. Available: http://ieeexplore.ieee.org/document/7387403/

[46] P. Gawlowicz, A. Zubow, M. Chwalisz, and A. Wolisz, "UniFlex: A framework for simplifying wireless network control," *IEEE International Conference on Communications*, 2017.

[47] C. Batista, P. V. Silva, E. Cavalcante, T. Batista, T. Barros, C. Takahashi, T. Cardoso, J. A. Neto, and R. Ribeiro, "A Middleware Environment for Developing Internet of Things Applications," in *Proceedings of the 5th Workshop on Middleware and Applications for the Internet of Things*. New York, NY, USA: ACM, Dec. 2018, pp. 41–46. [Online]. Available: https://dl.acm.org/doi/10.1145/3286719.3286728

[48] V. Gonzalez-Barbone, P. Belzarena, M. Randall, P. Romero, and M. Gelós, "GWN : A Framework for Packet Radio and Medium Access Control in GNU Radio," in *Wireless Innovation Forum Conference on Wireless Communications Technologies and Software Defined Radio (WInnComm 17)*, 2017.

[49] V. Gonzalez-Barbone, P. Belzarena, and F. Larroca, "Software Defined Radio: From Theory to Real World Communications," in *2018 XIII Technologies Applied to Electronics Teaching Conference (TAEE)*. IEEE, Jun. 2018, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/8476109/

[50] J. Goldsmith, C. Ramsay, D. Northcote, K. W. Barlee, L. H. Crockett, and R. W. Stewart, "Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoC Platform Using the PYNQ Framework," *IEEE Access*, vol. 8, pp. 129 012–129 031, 2020.

[51] A. Zubow, S. Rösler, P. Gawłowicz, and F. Dressler, "GrGym: When GNU Radio goes to (AI) Gym," in *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*. New York, NY, USA: ACM, Feb. 2021, pp. 8–14. [Online]. Available: https://dl.acm.org/doi/10.1145/3446382.3448358

[52] J. Mikulskis, J. K. Becker, S. Gvozdenovic, and D. Starobinski, "Snout – An Extensible IoT Pen-Testing Tool," New York, NY, USA, pp. 2529–2531, Nov. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3319535.3363248

[53] srsRAN, "srsran. your own mobile network," 2022. [Online]. Available: https://www.srslte.com/

[54] S. D. Eppinger, T. R. Browning, and J. Moses, *Design Structure Matrix Methods and Applications.*, ser. Engineering Systems Series. Cambridge: MIT Press, 2012.

[55] Osmocom, "osmocom Gnu Radio Blocks," 2016. [Online]. Available: https://osmocom.org/projects/gr-osmosdr/wiki/GrOsmoSDR

[56] The ZeroMQ Authors, "ØMQ: Documentation." [Online]. Available: https://zeromq.org/get-started/

[57] Z. Kang, R. Canady, A. Dubey, A. Gokhale, S. Shekhar, and M. Sedlacek, "A Study of Publish/Subscribe Middleware Under Different IoT Traffic Conditions," in *Proceedings of the International Workshop on Middleware and Applications for the Internet of Things*. New York, NY, USA: ACM, Dec. 2020, pp. 7–12. [Online]. Available: https://www.rti.com/products/connext-6https://dl.acm.org/doi/10.1145/3429881.3430109

[58] P. Hintjens, *ZeroMQ: Messaging for Many Applications*, 1st ed. O'Reilly Media, Inc, 2013.

[59] C. O'Connor, "StrictYAML," 2019. [Online]. Available: https://hitchdev.com/strictyaml/

[60] C. Yick, "Loading Dangerously: PyYAML and Safety by Design," 2019. [Online]. Available: https://www.serendipidata.com/posts/safe-api-design-and-pyyaml

[61] P. Biondi, "Scapy: interactive packet manipulation," 2003. [Online]. Available: https://scapy.net/conf/scapy_lsm2003.pdf

[62] The Python Software Foundation, "Entry points specification," 2013. [Online]. Available: https://packaging.python.org/specifications/entry-points/

[63] M. Fowler, *Refactoring - Improving the Design of Existing Code*. Addison Wesley Longman, Inc., 1999. [Online]. Available: https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Factory.html

[64] S. Gvozdenovic, J. K. Becker, J. Mikulskis, and D. Starobinski, "Multi-Protocol IoT Network Reconnaissance," in *Proceedings of the 10th annual IEEE Conference on Communications and Network Security (CNS)*, October 2022.

[65] B. Bloessl, C. Leitner, F. Dressler, and C. Sommer, "A GNU Radio-based IEEE 802.15.4 Testbed," in *12. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze (FGSN 2013)*, Cottbus, Germany, Sep. 2013, pp. 37–40.

[66] R. McNally, K. Yiu, D. Grove, and D. Gerhardy, "Fuzzing: The State of the Art," DSTO Defence Science and Technology Organisation, Edinburgh, South Australia, Tech. Rep., 2012.

[67] X. Jiao, "BTLE," 2014. [Online]. Available: https://github.com/JiaoXianjun/BTLE

[68] J. K. Becker, D. Li, and D. Starobinski, "Tracking anonymized bluetooth devices," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, pp. 50–65, 7 2019. [Online]. Available: https://doi.org/10.2478/popets-2019-0036

[69] The GNU Radio Project, "InstallingGR," 2021. [Online]. Available: https://wiki.gnuradio.org/index.php/InstallingGR

[70] Ettus Research, "USRP B200/B210 Datasheet." [Online]. Available: https://www.ettus.com/wp-content/uploads/2019/01/b200-b210_spec_sheet.pdf