

RFQuack: A Universal Hardware-Software Toolkit for Wireless Protocol (Security) Analysis and Research

Federico Maggi
Trend Micro Research
federico@maggi.cc

Andrea Guglielmini
Politecnico di Milano
andrea.guglielmini@mail.polimi.it

Abstract

Software-defined radios (SDRs) are indispensable for signal reconnaissance and physical-layer dissection, but despite we have advanced tools like Universal Radio Hacker, SDR-based approaches require substantial effort. Contrarily, RF dongles such as the popular Yard Stick One are easy to use and guarantee a deterministic physical-layer implementation. However, they're not very flexible, as each dongle is a static hardware system with a monolithic firmware.

We present RFQuack, an open-source tool and library firmware that combines the flexibility of a software-based approach with the determinism and performance of embedded RF frontends. RFQuack is based on a multi-radio hardware system with swappable RF frontends, and a firmware that exposes a uniform, hardware-agnostic API. RFQuack focuses on a structured firmware architecture that allows high- and low-level interaction with the RF frontends. It facilitates the development of host-side scripts and firmware plug-ins, to implement efficient data-processing pipelines or interactive protocols, thanks to the multi-radio support. RFQuack has an IPython shell and 9 firmware modules for: spectrum scanning, automatic carrier detection and bitrate estimation, headless operation with remote management, in-flight packet filtering and manipulation, MouseJack, and RollJam (as examples).

We used RFQuack to setup RF hacking contests, analyze industrial-grade devices and key fobs, on which we found and reported 11 vulnerabilities in their RF protocols.

1 Introduction

The increased adoption of wireless communication puts security research on the front line. Previous work has showed that both legacy [7] and newer-generation protocols (e.g., LoRaWAN [5]) require in-depth security auditing of the RF protocols. The impact of vulnerabilities on legacy protocols are particularly relevant, because they can affect industrial devices, which have long life spans (decades), and thus may never be patched until replacement.

After an almost-mandatory, blind replay-attack test with an SDR, the typical workflow to analyze an unknown wireless

protocol begins with a quick assessment using RF dongles (embedded RF transceivers), trying to sniff messages and "play around" with them. If the RF dongle fully supports the lower communication layers of our target protocol, then we have immediate access to the payload for further analysis. However, there exist many low-layer implementations, each with their own peculiarities, especially if we're looking at industrial applications. At this point we need an SDR to capture the signal and analyze it offline, with the goal of reverse engineering most of the lower communication layers. Once we have complete knowledge of the protocol, we can start looking for flaws. At this point it is not uncommon to bring RF dongles back in the game, because when the protocol is fully known, it's much more reliable to use a hardware transmitter to forge messages. Or maybe we need to quickly build a custom dongle, because we need a peculiar transceiver that supports a very uncommon modulation scheme.

Even if offline signal analysis is easier than in the past—thanks to advanced SDR tools like URH [11]—it's still quite challenging and error-prone to develop full, dynamic, precise, reliable transceivers. We wish we had flexible RF dongles that can be quickly reconfigured and adapted to support virtually any protocol, like with SDRs. This is what motivated us to develop and release RFQuack¹, which we like to think of as "the Arduino for RF researchers."

RFQuack is a RF dongle *system* with an extensible hardware and software platform that provides a solid foundation to develop custom RF dongles (see Figure 1) for wireless reverse engineering. While allowing full manual control, RFQuack includes ready-to-use RF-analysis features such as an automatic frequency and bitrate estimation that detects and clamps on transmissions in real time (in under 33 ms, with a 20 kHz accuracy). It supports multiple radios simultaneously, including mixed sub-GHz and 2.4GHz ones, and virtually any embedded radio. Also, RFQuack makes it easy to program interactive RF protocols without changing the firmware: It includes a packet filtering and modification engine that runs on the dongle and

¹<https://github.com/rfquack>

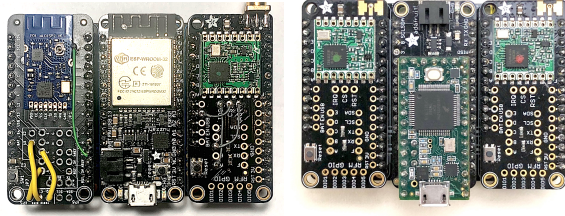


Figure 1: Two RFQuack modular dongles with main board (ESP32 and Teensy 3.2) and two RF daughterboards each (nRF24 + RF69 and two LoRa RF96, at 433 and 868 MHz).

can be scripted from the host.

We verified with practical use cases that the modular firmware makes it easy to implement new functionalities. This makes RFQuack suitable not only for security research, but also for hacking contests and trainings. In addition to an advanced IPython shell that leverages a robust Protobuf-based RPC, we believe that its open API will ease its integration with software such as URH, to offer a hybrid RF-analysis platform based on SDR and equally-flexible RF transceivers.

In summary, we make the following contributions:

- A complete and extensible open hardware and software system that makes hardware-assisted RF analysis flexible and easier to approach.
- We implement 9 firmware modules both to show how to use RFQuack’s API and to provide the essential functionalities to RF analysis and red-team tasks. Among these modules we include a signal-clamping routine that automatically decodes sub-GHz signals.
- Through a series of case studies on real consumer and industrial RF devices, we demonstrate RFQuack’s functionalities and provide a practical reference to its internals, to help future developers to extend it.

2 State of the Art and Motivation

The essential functionality of any RF-analysis system is to correctly “translate” a bit stream into its analog representation, by modulating the carrier waveform signal (or vice versa), and let users to easily “tap” into (or control) this process.

Currently available research tools can broadly be divided into SDRs and RF dongles. These approaches have opposite benefits and drawbacks, which motivated us in creating RFQuack to bridge this gap.

2.1 Software-defined Radios (SDRs)

To receive or transmit data with SDRs, we must implement most of the wireless protocol, down to the physical layer.

Hardware vs. Emulation. Traditional radio systems have hardware parts such as filters, converters, modulators, and demodulators, connected together to process any signal received by the antenna, and translate it into a different form

(e.g., audio)—and vice versa. In SDRs, such components are emulated in software, so they can implement any (wireless) communication stack without changing hardware.

High-end SDRs. While basic SDRs are essentially analog-to-digital converters (like the popular RTL2832U DVB-T receiver), professional SDRs integrate advanced digital signal processors (DSPs), field programmable gate arrays (FPGAs), and other dedicated, programmable hardware, that run the intensive tasks. The pure software part of such SDRs just configures the board and emulates the remainder components not available on the board.

Signal Sampling. Once tuned at a given frequency and bandwidth, an SDR board captures sample values of any received electric current, which is an analog, alternating signal (also known as *waveform*). These samples form a time series of complex values—often known as “I/Q data,” where I and Q are, respectively, the real and imaginary parts. SDRs capture (or generate) millions of samples per second (sps), producing a high-resolution digital representation of an analog signal waveform.

Post-processing. The received I/Q samples are transferred to the host via Ethernet or USB links for processing, with tools such as GNU Radio [3] or URH, which facilitate demodulating, interpreting, and decoding any (digital) data carried by the waveform. Regardless of the methodology, the outcome is a bit stream, which can be parsed for higher-level protocol dissection.

2.2 Embedded RF Dongles

To receive or transmit data with RF dongles, one must obtain or create a dongle that matches (i.e., implements) the physical layer of the target signal or device.

True Radios. RF dongles are small embedded systems with a host interface (typically serial, via USB or BLE), a microcontroller unit (MCU), and a digital transceiver module. The first and most-popular hacker-friendly RF dongle is the Yard Stick One [10], based on the Texas Instrument CC1111 chip, which integrates, within a single package, an MCU and the CC1101 sub-GHz radio module. The MCU runs a firmware that let host client software change RF parameters and transmit or receive data, by interacting with the registers on the MCU and the RF module. Unlike SDRs, RF dongles implement a protocol’s physical layer *in hardware*, which processes the analog signals received by the antenna into bit streams—and vice versa—exposing a digital communication interface to the host.

Decoding Capabilities. The capabilities of an RF dongle is bound to its transceiver, which cannot be changed since its soldered or integrated. For example, the Yard Stick One, makes it straightforward to decode data modulated with ASK or FSK, but PSK is simply not supported by the CC1101. Unlike with SDRs, the analyst must know the RF parameters

Table 1: Feature comparison of SDR-based vs. RF-dongle-based approaches and tools, discussed in Section 2.3

	SDR		RF Dongles		
	GNU Radio	URH	Single Radio	Multi Radio	RFQuack
Spectrum Coverage	10 ³ to 10 ⁶ Hz, continuous		Only specific bands		Arbitrary bands, discrete
RF Parameters	Any (custom physical layer)		Hardware bound		All (modular)
Host Interfaces	Wired (USB, Ethernet)		USB, BLE	USB, BLE, WiFi, Cellular	Any (modular)
API Uniformity	High (established project)		Very low		High
Extensibility	Very high	High	Very low (SW)		High (HW, SW)
Main Target Layers	Physical	Logic	Logic	Hybrid	Hybrid
Focus	Research	Research, RE, Fuzzing	Red-teaming	Red-teaming	Research, RE, Fuzzing
Development Effort	High (DSP)	Medium (UI, Scripting)	Low (Scripting)	Low (UI, Scripting)	Medium (Scripting, C)
Transceiver Performance	Medium (Latency)		Very high (IC)	High (SPI)	High (SPI)
Hardware cost	\$20–15,000		\$10-200		

(i.e., modulation scheme, carrier frequency, and bitrate) *a priori*, as the (de)modulation happens in hardware.

Post-processing. The firmware running on an RF dongle "talks" with a companion client tool on the host, typically a mobile application or command-line interface (CLI) that makes the dongle actually usable. RfCat [2] is the first and most-popular CLI for CC1111-based RF dongles such as the Yard Stick One or the PandwaRF [12]. Without going into further details, there exist similar solutions for the 2.4GHz bands: The nRF24 is a popular hacker-friendly radio transceiver, which has inspired the creation of various RF dongles, firmware, and utilities (e.g., nRF Research Firmware [8]).

2.3 Motivation: The Best of Both Worlds

The diverse characteristics of SDRs vs. RF dongles summarized in Table 1 have motivated researchers and hardware developers to create feature-rich boards with multiple transceivers that cover the most common bands. For instance, the HackCube integrates an SDR receiver, WiFi, Bluetooth, NFC, and a 2.4GHz and sub-GHz transceiver. While tools such as the HackCube have merit, they are still monolithic (for miniaturization reasons). Without an open, flexible design, it's hard for the community to maintain and contribute to the development of such devices: Adding hardware or software modules to devices like the HackCube is possible by only patching it, as it's not designed for extensibility.

The framed text in this section indicates an observation or design principle beneath RFQuack.

Spectrum Coverage and RF Parameters. The main characteristic of SDRs is their wide spectrum coverage (or bandwidth). Even a \$20 SDR can tune at any frequency between 300MHz and 1.5GHz. By design, RF dongles can only tune to a very limited, discrete set of bands (e.g., 125kHz, 315MHz, 433MHz, 2.4GHz) because their RF frontends have a basic frequency synthesizer.

Since the physical layer is implemented in hardware, RF dongles can only support a finite set of modulation schemes, synchronization words, packet formats, and CRC algorithms.

Adding new frontends for RF dongles to support new bands and physical layers should be as easy as connecting the right hardware module, like in advanced SDR systems (e.g., Ettus USRP), which come with a variety of daughterboards.

Host Interfaces. SDRs need high-bandwidth interfaces such as USB or Ethernet. The most popular RF dongle (Yard Stick One) is based on USB, while the more recent PandwaRF, WHID Elite, or HackCube, have BLE, WiFi or cellular interfaces, mostly because they are conceived with a red-teaming purpose (e.g., physical security), so they can launch attacks remotely, in a headless fashion.

If RF the many dongles had a uniform connectivity layer on top of such interfaces—instead of custom, undocumented protocols—researchers would be empowered to develop with these platforms and benefit from their diverse features.

API Uniformity. GNU Radio offers a uniform, documented API. While performance-critical signal-processing blocks can be written in C++, GNU Radio allows to develop radio applications in Python, and compose them with a GUI. URH exposes a plugin API to extend its signal-processing, decoding, and fuzzing capabilities.

The only effort to provide a uniform API in the RF dongles world is PandwaRF's Android API and SDK. However, being the PandwaRF a commercial, single-radio product, its firmware is not open source and the hardware capabilities are bound to the fixed RF frontend. Similar efforts should be embraced by open source RF dongle systems.

Extensibility and Development Effort. Research tools like URH make RF reverse engineering accessible, although implementing complex protocols is challenging: Despite GNU Radio offers a high-level experience with fine-grained control over the processing pipeline, RF protocol design calls for in-depth DSP knowledge.

While it is easy to extend and port software across different SDR hardware thanks to the uniform API exposed by toolkits like GNU Radio or URH, the RF world is very different: Each system has its own custom firmware. As a result, even when

the firmware is open sourced, it is hard to extend unless by patching it.

Extending the firmware of an RF dongle should be facilitated by a consistent API, which allows the developers to focus on adding new functionalities, rather than reinventing custom abstraction layers.

Main Target Layers. SDRs allow to work at any protocol layer. Although tools like GNU Radio are meant primarily to work at the physical layer, they have support to parse the logic of digital protocols (e.g., packet format). Instead, URH is focused on supporting the reverse engineering of the *logic* layer.

RF dongles only allow to interact with the logic layer of protocols. Depending on the RF frontend, some RF dongles may allow partial access to lower layers. For instance, some RF dongles allow to consume RSSI² samples fast enough to estimate the spectrum density. In Section 3.4 we show how we use these samples to implement the auto-tuning feature of RFQuack. The HackCube even includes a simple SDR receiver, although creating custom receivers is still constrained by the MCU computational power.

Focus. Both SDRs and RF dongles can be used for research purposes, but there are some small differences. Out of the box, GNU Radio is less fit for offensive tasks, although it does not prevent to develop custom, even very advanced, physical-layer fuzzing routines. URH is conceived with a clear RE orientation: Besides the UI that assists protocol parsing, URH has ready-to-use functions to create fuzzing templates and them directly against the targets.

RF dongles are optimized for offensive and red-team activities. For example, the PandwaRF has rolling-code-cracking routines, WHID Elite [4] comes with a modem (to simulate remote attackers) and a USB port (for HID exploits).

To draw an analogy, RFQuack is more inspired by URH than by the available RF dongles: It facilitates the extension of its firmware to support custom protocol parsing, complex tasks such as fuzzing or interactive logics, and to test for known vulnerabilities (e.g., RollJam, MouseJack).

Development Effort. It is not only hard to develop physical-layer protocols (e.g., with GNU Radio), but it requires in-depth knowledge in analog and digital signal processing, information theory, and cryptography. Frameworks like GNU Radio are very sophisticated and focused on technical advances rather than on *user* documentation and usability, which make them not easy to approach. URH has made a step forward, by providing a modern UI that works out of the box, a plug-in API, and scripting capabilities. RF dongles make "quick" tasks very easy to approach, as they offer a simple UI and Python scripting, but it is not immediate to develop

²The receiver signal strength indicator is the measured power of a received signal at a given carrier frequency.

firmware extensions, and nearly impossible to change their hardware (most RF modules are integrated or embedded on the board).

Transceiver Performance. Single-radio RF dongles can embed the radio frontend and the MCU in a single, integrated SoC package, which minimizes latency on the bus. While SDRs can be optimized with FPGA or hardware DSPs, a non-negligible portion of the protocol consists of native or interpreted (Python) code running on the host computer, on top of a general-purpose OS, which cannot comply with the determinism and real-time constraints required to implement accurate, efficient, and reliable transceivers.

While RFQuack's design imposes some latency due to the detachable RF frontends, the physical layer is implemented in hardware, which guarantees far more precision and reliability than any SDR implementation. Some protocols—especially in the 2.4GHz bands and above—simply cannot be conveniently developed with an SDR, if not with very expensive hardware and FPGA offloading.

3 RFQuack High-level Overview

Both hardware and software of RFQuack are fully modular. This ensures that we can always leave convenient "tapping" points for users to customize, while relying on a uniform interface to multiple radio transceivers.

In addition to using the built-in IPython shell, we make it easy for users to build firmware modules to implement signal-reconnaissance tasks, transmit or receive (post-processed) data, discard packets not matching filters (e.g., selective protocol sniffing), as well as interacting at the low level with the MCU and the transceiver (e.g., read and write registers).

3.1 The Journey of a Wireless Transmission

Reading Figure 2 from left to right, (1) when a target device (e.g., an IoT embedded system) transmits packets, the signal is received by one or more of the daughterboard radios. An interrupt request from daughterboard (2) invokes `onPacketReceived(pkt)` of all loaded module, in a user-configurable order. For example, if enabled, (3) the packet-filtering module ignores packets not matching a set of user-provided regular-expression patterns, and enqueues the accepted ones for further processing. If enabled, (4) the packet-manipulation module pulls packets from such queue, (5) pipes them through a set of user-provided modification rules, and (6) enqueues them again. The repeater module, if enabled, (7) consumes any enqueued packet and (8) pushes them to another queue. Depending on the settings, (9) packets can be serialized and proceed towards the host. Alternatively, the user may set RFQuack to (11) forward the (modified) packets to the repeater module, which immediately transmits them.

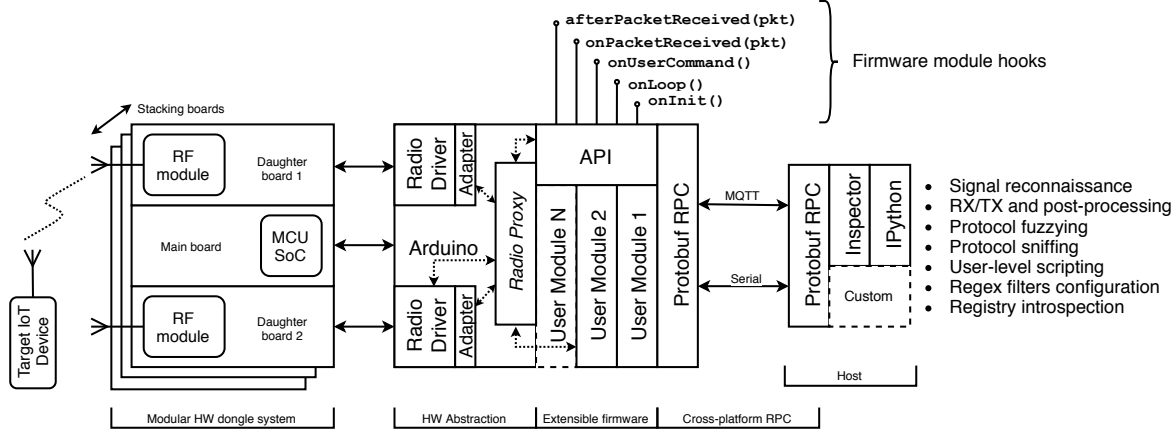


Figure 2: The high-level architecture of RFQuack presented in Section 3.1 with details in Section 4.

3.2 Modular Hardware Design

It is possible to swap RFQuack daughterboards because they’re connected with simple pin headers. Multiple boards, with up to 4 RF daughterboards each, can be stacked using a simple pin header. Each daughterboard needs a serial peripheral interface (SPI) bus, plus some optional GPIO pins. The 3 SPI bus lines (SI, SO, CLK) are shared across all daughterboards (SPI slaves). To allow the MCU (SPI master), to selectively talk to a specific transceiver, each RF daughterboard independently routes 1 slave-select (SS) line to the main board, plus 1 or more GPIO lines (e.g., for interrupts). The MCU triggers the SS to notify a specific transceiver that there is data on the SPI bus. The transceivers can start a SPI transaction by triggering an interrupt on a GPIO.

3.3 Modular Firmware Design

As shown Figure 2, the module API exposes 5 methods, which allow to implement user modules that can tap into the data-processing pipeline: `onInit()`, when the module is first loaded; `onLoop()`, at each MCU cycle; `onUserCommand()`, when a valid command is received via the RPC; `on/afterPacketReceived(pkt)`, called upon and after a packet is received from the transceiver. Each functionality of RFQuack is implemented as a module. For example, the radio-management module receives and dispatches users commands (`onUserCommand()`) such as set/get registers and set/get RF parameters. The packet-filtering module, instead, is triggered on the `onPacketReceived(pkt)` event, and calls `onLoop()` to flush any received or manipulated packet as soon as possible. In Section 5.5 and 5.4 we showcase the module API by implementing RollJam and MouseJack, two popular RF attacks.

Transceiver-agnostic, Uniform API. Like with GNU Radio sink blocks RFQuack’s abstracts different transceivers via the same API, which provides functions to change the radio mode (RX, TX, idle, jam, promiscuous), RF parameters (carrier, bitrate, deviation, bandwidth), packet format, output power, to get-set registers, and to send-receive binary data.

The list of exposed functions is in Appendix B. Optionally, it exposes transceiver-specific functions (e.g., the CC1120 has 16 bit extended registers).

Connectivity and Headless Operation. To meet red-teaming operational requirements—although more compact RF dongles exist—RFQuack is designed with an open connectivity model in mind. RFQuack has built-in support for serial (USB) and MQTT transport (for remote, distributed setups), and the client side supports deserialization and automatic type inference.

3.4 Automatic Signal Clamping

In addition to the precise carrier frequency of the target communication, we must know its data rate (or bitrate, bits/sec) to correctly interpret a demodulated signal into bits. Usually, when short of luck at inspecting FCC (leaked) documents, we employ an SDR and a spectrum analyzer to measure the power value at each frequency, and visually identify the highest peak. Similarly, if the bitrate is unknown, the only option available when using RF dongles is to set the bitrate to the highest value, set the radio in promiscuous mode (i.e., do not filter incoming packets based on preamble length or sync-word), and post-process any captured data to downsample it to the correct bitrate. This is not ideal, because it does not allow the user to fully leverage on-board filters and other functionalities of the firmware.

RFQuack has a **frequency-finder module** that listens on a range and, as soon as it detects a new transmission, tunes to that frequency and triggers the **bitrate-estimator module**. For example, if a car’s key fob is pressed within range, the modules infers carrier frequency and bitrate, and present the user with the payload. This happens in real time, entirely on the dongle, without interfering with other tasks. It is important to clamp on the signal before its preamble ends, so that RFQuack will be able to synchronize and decode the transmitted payload. Although RFQuack supports multiple radios, these modules uses a single transceiver, leaving any spare transceiver available to work on other frequencies.

While the PandwaRF supports similar functionalities, we’re the first to implement it as a module and, most importantly, release it as open source code.

4 Implementation Details

RFQuack modular hardware is based on Adafruit’s Feather system, as it comes with stacking boards with up to 4 slots each. We tested early versions of RFQuack on the ESP32 and ESP8266 SoCs (MCU + WiFi), the RFM69 and CC1120 RF modules (sub-GHz bands), and the SIM800 cellular modem. We provide the open-hardware schematics for a CC1120 adapter and a compact nRF24 FeatherWing. The most recent version supports the CC1101 and nRF24 RF modules³, working simultaneously, with up to 5 radios (a shared SPI bus plus 2 GPIOs are required for each radio), and we tested it with the Teensy 3.2. Thanks to the Arduino and similar abstraction frameworks, RFQuack could run on virtually any of the 800 boards supported by PlatformIO⁴.

4.1 Loop, Dataflow and Queues

RFQuack’s firmware runs a high- and a low-priority loop. Any new packet goes through all the high priority tasks and then, via a decoupling queue, through the low priority ones. Each module may implement different—high or low priority—hooks to interact with the underlying framework.

For better compatibility, we do not to leverage hardware parallelism, since it is supported only by few boards (e.g., ESP32 has two cores, while ESP8266 is single core). However, if a user wants to trade off compatibility for speed, they can just change RFQuack’s main `loop()` to assign the high- and low-priority loops to two distinct cores.

4.2 Multi-radio Abstraction Layer

RFQuack abstracts each RF frontend and exposes a *proxy* API. Instead of writing our own native drivers for each RF frontend, we use a decoupling driver *adapter* that wraps the native driver. We treat native drivers as external dependencies (we mainly use RadioLib [1]), which can evolve independently from RFQuack. As RFQuack supports multiple radios simultaneously, the proxy forwards each request to the correct driver. For example, `setModulation(OOK, RadioB)` sets the modulation of the second frontend to OOK. Since each RF frontend may have unique features, not covered by the proxy API, we still allow direct access to the native driver.

4.3 Carrier Frequency Detection

Many transceivers provide the RSSI value, an estimate of the power received, in decibel. Samples of the RSSI can be used to draw a low-resolution spectrum (e.g., the PandwaRF Android mobile application provides that) and detect peaks, in order to spot transmitting devices nearby. A naïve approach would

be to simply loop through all the frequencies in a target range and collect RSSI samples. We measured that it takes no less than $1ms$ for the CC1101 to tune and provide a reliable RSSI sample, which means at least $5s$ to scan the 432–437MHz range with a kHz resolution.

Scan by Region. Instead of tuning to all the frequencies, we use a scan-by-region approach, leveraging the programmable receiver bandwidth filter available on any modern transceiver, which let us narrow the range of frequencies that influence the RSSI.

We set the filter bandwidth to the maximum value, B_{max} , take RSSI samples in the middle of each of the N regions, and identify the most active one.

We tune to the center, f_i , of each region, starting from a given offset, f_o , is then:

$$f_i = f_o + i \cdot \frac{(1-c)}{2} \cdot B_{max} \quad i \in \{0, 1, 2, \dots, N\},$$

where $c \in (0, 1)$ is the region-overlap ratio. As shown in Figure 3, the regions must overlap to avoid corner cases due to the natural attenuation of the non-ideal bandpass filters. A guiding criterion to set the value of c is that it should be proportional to the filter slope, which can be obtained from the filter response reported on the transceiver datasheet. We obtained reliable results by setting $c = 0.25$ on the CC1101.

Trichotomic Search. We run a fine-grained search within the most active region to find the peak frequency, by halving the receiver’s bandwidth at each iteration. To avoid ties while keeping efficiency, after halving the bandwidth, we split the resulting search domain in 3 sub-regions. This narrows down the theoretical search time—within a 5MHz range—to $21ms$, while a linear search of each of the 87 regions⁵ would take $87ms$.

Tuning Time Optimization. To limit the time t_{tune} required by the transceiver to tune and provide stable RSSI readings, we pre-compute and cache the calibration registry values.

Most transceivers include self-calibration routines, which must be run before tuning to a frequency (or channel). For instance, the CC1101 can hop to a frequency in $t_{hop} = 75\mu s$ and calibrate in $t_{cal} = 712\mu s$. The radio driver also introduces a latency when sending the calculated registry values on the SPI bus. We instrumented the firmware, tuned the radio 100 times on different frequencies and measured $t_{driver} = 320 \pm 20\mu s$.

Since the range of frequencies is known in advance, we precompute and cache the calibration registry values for each frequency. Overall, $t_{tune} = t_{hop} + t_{cal} + t_{driver} + t_{RSSI} = 75 + 712 + 320 + 600ms \leq 1ms$, where t_{RSSI} is the minimum time for the radio to provide a stable RSSI value.

³We’re currently porting from RadioLib the wrapping code to support the RFM2x, RFM69, and RFM9x LoRa modules.

⁴<https://platformio.org/boards>

⁵The widest receiver passband filter on the CC1101 is 812kHz, which means 9 regions for the first pass and 87 iterations with progressively narrower filters, down to 58kHz.

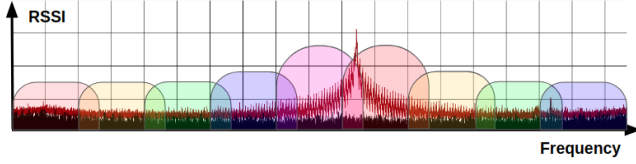


Figure 3: Carrier frequency peak detection (Section 4.3).

4.4 Automatic Bitrate Estimation

By default, RFQuack sets the receiver to its highest supported bitrate, thus oversampling any incoming transmission. The Nyquist–Shannon sampling theorem implies that an analog signal must be sampled at least twice its bitrate to reconstruct the rectangular wave⁶. Any digital signal (especially in the sub-GHz bands) always starts with a preamble (i.e., a sequence of alternating 0s and 1s) to "wake up" the receiver, which will use it to synchronize to the incoming packet. At this point, as soon as RFQuack has received enough over-sampled alternating symbols with repetitions (e.g., four 1s followed by four 0s, repeated 3 times), it uses this array of symbols to estimate the true bitrate. The estimated bitrate, \hat{r} , is such that the received array (oversampled at r_o) has only one repetition of each symbol. We approximate this with the (weighted) average of the number of consecutive 1s (or 0s) found in the preamble P :

$$\hat{r} = \frac{r_o \sum_i w_i}{\sum_i |p_i| \cdot w_i} \quad (1)$$

Where $|p_i|$ is the number of consecutive 1s within the preamble P and w_i is the number of occurrences of p_i in P . For example, $P = \bar{1}00\bar{1}\bar{1}\bar{1}000\bar{1}\bar{1}000\bar{1}\bar{1}\bar{1}$ counts as $1 \cdot 1 + 3 \cdot 2 + 2 \cdot 1$ divided by the sum of weights, $1 + 2 + 1$. Since the preamble does not carry any information, we accept to lose it during this operation, with the advantage of not requiring to store the packet for downsampling: We adjust the correct bitrate *before* the preamble ends, so any subsequent incoming data is captured at the correct bitrate.

For simplicity, we assumed a 2-symbols OOK scheme: Section 7 explains how to extend this to other modulation schemes. The 2.4GHz is very different and this functionality is not really needed: The nRF24 transceiver, and most of the 2.4GHz transceivers, only support 2–3, very high, bitrate values.

Runtime Estimation. The only operation run on the MCU is a loop over a fixed-length bit stream, P , to count the repetitions in p_i . The upper bound is determined by the chosen oversampling bitrate, r_o , and minimum expected preamble length, after which an estimation is available.

For example, to capture signals up to 15 kbps, we set $r_o = 30\text{kbps}$ and stop after 32 bytes (half the size of the CC1101's FIFO), which translates into roughly $8.53\text{ms} = \frac{1}{30\text{kbps}} \cdot 32 \cdot 8\text{bits}$.

⁶This limits *any* bitrate-estimation technique to half of the receiver's maximum bitrate.

Table 2: Key fobs used in our first case study.

Car Model	Key fob Model	Freq. (MHz)	Bitrate (kbps)	Transm. len. (ms)
Fiat	RX2TRF198	433.92	9.6	295
Nissan	TWB1G766	433.92	5.0	370
Audi	HELLA FS12A70	434.42	3.4	183

5 Real-World Case Studies

The goal of this section is to showcase how RFQuack applies to various scenarios, from fuzzing unknown protocols to creating custom modules that implement (known) exploits. We have presented some these case studies as PoCs of RFQuack at the CanSecWest conference and at the Armory of Hack In The Box.

5.1 Sniffing Key Fobs and Opening a Car

We started with cars key fobs, since they are widely available RF transmitters, they operate on the ISM bands, they are interesting targets (since they protect valuable assets from theft and robbery), and transmit a rolling code, so the payload content changes while the structure is fixed. We used the 3 key fobs listed in Table 2. Also, we used a BladeRF to capture the signals emitted by these transmitters and used them as a baseline to validate the auto-clamping modules (see Section 6).

Using the IPython shell connected via serial to a RFQuack dongle, we configured the system in promiscuous mode. Despite the RF noise, and even if we didn't know the sync word in advance, we quickly setup the packet-filtering module to discard any packet without a valid preamble. Using the console it was easy to transmit some packets back to the receiver, which were discarded because of the rolling-code mechanism⁷. It was during this experiment that we discovered that the receiver of one the cars was vulnerable to replay attack⁸: We were able to deterministically open the doors despite the code was rolling at any transmission.

5.2 Vulnerable Industrial Devices

We analyzed the protocol of 5 sub-GHz industrial radio devices used to automate and control manufacturing and logistic processes⁹.

In addition to sniffing packets for analysis, we used RFQuack to implement a loop to find the correct rolling code, given a rolling code we knew from another, same-vendor device.

```
q.packet_filter.add(pattern="^aaaa", negate=False)
q.radioA.rx(); q.radioB.tx()

for pkt in q.data:
    # we knew rcode0 from another, same-vendor device
```

⁷Video of this demo: <https://youtu.be/jLI-oby2mu0>

⁸We're reaching out to the car manufacturer for responsible disclosure, because the car model and brand is current and very popular.

⁹We cannot explicitly mention the vendor names without indirectly revealing our names, because we are the only research group that looked at those devices and there was quite extensive media coverage. In case of acceptance we will include all the identifying details.

```
pkt[2:4] ^= rcode0 # XOR with known rolling code
q.radioB.send(pkt) # transmit
```

Once we heard the receiver accepting our command, we knew that we had used the correct rolling code. This allowed us to exhaustively build a table of valid rolling codes. Knowing valid rolling codes and the packet structure, we configured the filter, manipulator, and repeater modules of RFQuack to wait for one valid transmission, change some specific bytes in the packet, and transmit the modified packet.

```
q.packet_filter.add(pattern="^aaaa", negate=False)
q.radioA.reset_packet_mods() # reset module

# XOR byte 7 with 0x04, byte 10 with 0x08, etc.
q.radioA.add_packet_mod(i=7, val=0x04, op=XOR)
q.radioA.add_packet_mod(i=10, val=0x08, op=XOR)
q.radioA.add_packet_mod(i=12, val=0x04 + 0x08, op=XOR)
```

As a result, the vulnerable target receiver executed a command of our choice. Moreover, since the routine was reactively executing the same loop in a headless and continuous fashion, we were able to keep the target in to a persistent denial-of-service state, by repeatedly sending "shut down" commands. Note that the packet-modification routine runs on the MCU, and could be used to implement multi-radio scenarios like the one just presented.

5.3 Sniffing 2.4GHz Protocols

The two most challenging aspects of 2.4GHz protocols are that the spectrum is very crowded (e.g., cellular, WiFi, Bluetooth) and protocols can use frequency hopping. This makes a pure SDR approach quite impractical. We conducted this experiment outside a Faraday cage, in challenging conditions: In addition to cellular traffic, there was a WiFi AP and a Bluetooth smartwatch. Knowing only the bitrate from the public FCC database, we were able to identify the exact frequency of a Microsoft wireless mouse and narrow down its address in seconds, with only 6 minutes of manual work on the command line (see Appendix D.3).

By generating several packets (i.e., by moving the mouse excessively nearby the receiver) we isolated its sync word (and confirmed it from FCC database)¹⁰. Then, by looping once again through all the frequencies, we isolated only the traffic coming from that mouse, given that we knew the sync word. The same experiment would have required hours of SDR development.

5.4 MouseJack Attack Implementation

MouseJack is a well-known attack against non-Bluetooth 2.4GHz HID devices such as mice and keyboards, which has recently enjoyed quite some attention [9], because many vulnerable, unpatched devices are still sold and used. We have taken this popular attack as a representative example to show how to create a custom user module that implements the

exploit. The disarmed source code (i.e., without payload) is available at RFQuack's repository, so we hereby focus on how the implementation leveraged RFQuack's module API.

The MouseJack module overrides the `onUserCommand()` to react on "start/stop" commands sent over serial or MQTT, to receive the attack payload (e.g., HID commands) from the user. The "start" command puts the radio in promiscuous mode (using the built-in `setPromiscuousMode()` method) and starts receiving (`setMode(RX)` and `setFrequency(2400)`).

The MouseJack module overrides the `onPacketReceived(pkt)` function, to check if the scanning loop has captured a valid payload, by checking that the packets contain a valid sync-word and pass the CRC. Based on the received valid packets, the module continues and fingerprints the transmitting victim device (e.g., Microsoft, Logitech). The resulting module has under 350 lines of code, while the reference standalone implementation has over 470 lines, plus libraries.

5.5 RollJam Attack Implementation

In the sub-GHz range, RollJam is a very popular rolling-code-cracking attack. The inventor [6] used a custom-made device with two transceiver: One is to jam the legitimate receiver (i.e., on the car), and the second to listen for packets from the key fob, to capture rolling codes—which will never be received by the legitimate, jammed receiver. Last, RollJam stops the jamming loop and transmits the captured still-fresh rolling code, which will be accepted by the receiver that couldn't see such code while blinded. We have taken this popular attack as a second, representative example to show how to develop custom modules by leveraging RFQuack's firmware API to multiple radios. The source code is available at RFQuack's repository, so we hereby focus on how the implementation leveraged RFQuack's module API.

Totalling 139 lines of code, the RollJam module implements the `onUserCommand()` function to handle "start/stop" commands, and to allow the user to configure the number of packet repetition, and which radio to use for jamming and which one for listening. The "start" command puts the listening radio in receive mode (`setMode(RX, listenRadio)`) and calls the `setMode(JAM, jamRadio)` on the jamming radio. Any valid incoming packets will trigger the `onPacketReceived(pkg)` function, implemented by the module. If enough valid data is received, the jamming radio is idled and the listening radio is used to repeat the last valid rolling code (`listRadio->transmit(pkt)`).

5.6 RF Hacking Contests and Trainings

Capture the flag (CTF) competitions have fully embraced RF hacking challenges, to the point that we can participate to RF-only contests, with Capture the Signal¹¹ and Hack-a-Sat¹² being the most recent ones.

¹⁰Video of this demo: <https://youtu.be/c40Sh3jQN5Y>

¹¹<https://cts.ninja>

¹²<https://www.hackasat.com/>

While reverse engineering offline signal captures is a useful learning experience, real-world targets will always be physical, interactive devices with real radio transceivers. RFQuack makes it easy to provision interactive devices thanks to its scripting capabilities and firmware modularity, while maintaining fine-grained control on the RF protocol, which is an essential aspect for contests and trainings, because they require progressively more difficult challenges.

For example, we used RFQuack to build a remotely-programmable, battery-powered beaconing device that we hid in a conference room, activated it during a presentation, and asked the attendees to start inspecting the spectrum "around a certain frequency". We asked them to locate the beacon, on which we physically printed a secret key, which they to transmit to the same device to unlock the next-level challenge.

Thanks to its cross-platform backend protocols (e.g., MQTT, including AWS IoT's MQTT), it is very easy to integrate RFQuack nodes into a web service, for example to emulate a complex IoT device with a web frontend. RFQuack's firmware is already delivered via automated, "dockerized" builds, which make it suitable for automated, scalable deployments on multiple, distributed nodes.

6 Experimental Evaluation

While that the most useful aspect of RFQuack is its flexibility, the system must be robust in order to be usable. Our first goal was to show that RFQuack can automatically capture sub-GHz transmissions in real time. Then, we investigated the accuracy the automatic signal clamping feature, so that users know what to expect from it.

6.1 End-to-end Experiment

We used a key fob to generate a reference signal, which we captured with a BladeRF SDR and decoded it with URH to know the baseline RF parameters. Among the key fobs in Table 2, we choose the Audi's HELLA FS12A70, because it transmits the shortest packet, which will put our auto-clamping module under time pressure¹³.

We pressed the key fob button 50 times while the dongle was actively searching for valid transmissions. We used a regular expression to parse the log and extracted the running time for both the frequency finder and the bitrate-estimation modules, and the recovered frequency and bitrate values. We correctly identified and decoded 43 over 50 transmitted packets, which is a very positive result outside a Faraday cage. The remaining 7 packets were wrongly decoded due to wrongly inferred parameters.

Results. The results in Table 3 are aligned to our theoretical estimations. The measured time for frequency estimation is about 1.5 ms greater than the expected. We traced this back to the native radio driver performing some additional SPI

¹³Video demo using the frequency and bitrate recovery on various key fobs: https://youtu.be/36Bt8un_Y-Y

transactions to handle special configurations. For example, every time a packet is received, the driver queries the radio for the CRC configuration. We optimized some of these transactions and we plan to review the rest of the native drivers for optimization. The overall running time is ≤ 33 ms, which is shorter than the preamble.

The mean values are affected by the 7 incorrect estimations. We experimentally validated that a standard deviation up to 0.2 MHz is not posing issues to the decoding phase. Thus any estimated frequency between 434.2 and 434.6 MHz is considered valid.

The estimated mean bitrate is affected by outliers because this step runs after the frequency estimation. For this reason, an incorrectly estimated frequency almost always implies a wrong bitrate value. This explains the high standard deviation value despite the accurate mode. Moreover, we experimentally validated that a bitrate deviation up to 0.3 kbps does not affect the decoded signal.

6.2 Frequency Recovery Accuracy

We first focused on the frequency recovery part, using the second embedded radio to transmit the reference signal. Although the transmitted payload is irrelevant for this experiment, we used the packet of one of our key fobs in order to have something realistic in terms of preamble length, syncword, and total packet length.

To run the experiment we wrote a small script (see Appendix D.1) that tunes both radios at 432–437 Mhz, transmits with the first radio and collects any packets received by the second radio, along with the exact, detected carrier frequency.

Results. The results shown in Figure 4 (left) show that the algorithm is able to detect the frequency accurately, with an average error of 0.02 MHz, and an average standard deviation of 30 kHz. This result is in line with our expectation since the narrowest receiver filter bandwidth for the CC1101 is 58 kHz. It follows that the error decreases while approaching the center of the receiver bandwidth, and increases up to 30 kHz when moving away from it. Note that a 30 kHz error is acceptable: Recall that the RollJam attack works by jamming the receiver at 50 kHz off the exact tuning frequency.

6.3 Bitrate Recovery Accuracy

We used once again a script (see Appendix D.2) to automate the TX/RX of packets from 1 to 15 kbps. Following Nyquist–Shannon theorem, we oversampled at 30 and

	t_{freq} [ms]	t_{br} [ms]	$Freq$ [MHz]	Br [kbps]
Baseline			434.42	3.40
Mean	22.55	10.26	434.48	3.80
Mode	22.56	10.21	434.45	3.43
First Quartile	22.53	10.19	434.42	3.38
Third Quartile	22.57	10.35	434.46	3.45
Standard Deviation	0.08	0.13	0.10	1.52

Table 3: Speed and accuracy on key fobs (Section 6.1).

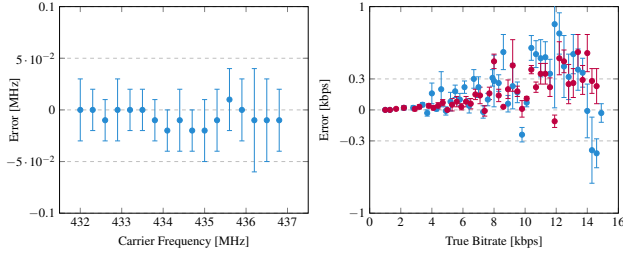


Figure 4: Carrier frequency peak detection and bitrate recovery accuracy (oversampling at 30kbps and 60kbps).

60 kbps, so obtaining the data plotted in Figure 4 (right).

Results. We obtained good precision at low bitrates and a slightly higher precision overall at 60 kbps, as expected. We manually validated that the CC1101 can successfully decode signal with up to 0.3 kbps estimation error. Therefore, oversampling at 60 kbps guarantees no data loss, up to 15 kbps.

The reason why it almost seems that Nyquist-Shannon doesn't hold is twofold. First, from Equation (1) recall that shorter sequences of consecutive 1s (or 0s) have greater influence on the estimated bitrate. The limit is exactly the Nyquist frequency, where the number of consecutive symbols tends to 1, determining wide oscillations of the estimated value. The greater is the resolution, the less influence a single error has.

Secondly, after clamping to a signal, most transceivers (including the CC1101) internally estimate the clock from the incoming symbols, using a re-synchronization routine, which run continuously to eliminate small discrepancies between the chosen and real bitrate. Thus, if oversampling bitrate is not an integer multiple of the incoming signal bitrate, the transceiver will automatically try to adjust the bitrate to compensate for unexpected symbols. This feature, which cannot be disabled, influences our estimation. We were not aware of such feature and we will search if it can be disabled in some transceivers.

7 Limitations and Future Work

Like most community-driven projects, RFQuack also is meant to be a complete, final solution. We focused on providing solid foundations and a flexible firmware for people to build upon. There are, however, some limitations that we think should be addressed in the near future.

Scanned Bandwidth Limit. The first phase of the frequency recovery algorithm loops through all regions in the configured band. It's still far more efficient than looping on the single frequencies, because it splits the spectrum in to a number of regions, thus reducing the number of iterations, but if the range is very wide, the running time may be incompatible with an online signal capture scenario. This is a technical limitation, so the easy solution is to just use a transceiver with a wider filtering bandwidth. Alternatively, the algorithm could be extended to shard the band across multiple transceivers.

Better Modulation Guessing. The bitrate estimation algorithm assumes OOK modulation. To support other modula-

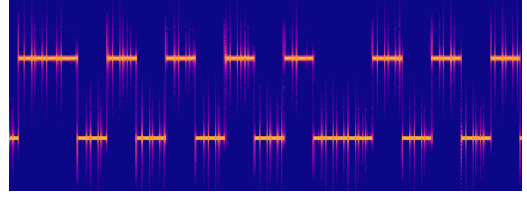


Figure 5: A single 2-FSK transmission, or two complementary OOK transmissions.

tions such as 2- or 4-FSK, we foresee two solutions. The frequency finder should look for the strongest RSSI on N frequencies, to infer the number of occupied frequencies. For example, a single occupied frequency may denote OOK, while 2 or 4 occupied frequencies indicate 2- or 4-FSK.

Alternatively, one can demodulate 2-FSK as if it was OOK, by using offset tuning. FSK encodes symbols as discrete frequency changes, but this can be seen as two independent, complementary OOK transmissions on two distinct frequencies, as shown in Figure 5. Thus, once one of the two peak frequencies is found, the algorithm should tune "on the side" of just a few kHz and set the narrowest filter bandwidth in order to "see" only one of the two transmissions. A second radio can be used to parallelize this task.

8 Conclusions

We presented the first truly modular RF system for security analysis of RF protocols. RFQuack is midway between SDRs and RF dongles, by providing uniform high-level APIs to reconfigure an easy-to-customize hardware system, as well as the advanced features typically found in RF dongles such as the Yard Stick One.

We hope that it will encourage the community to implement new features, as we think that RFQuack can really change how we approach RF research and training.

References

- [1] Universal wireless communication library for Arduino. "RadioLib." URL: <https://github.com/jgromes/RadioLib>.
- [2] Atlas. "RfCat - swiss-army knife of ISM band radio." URL: <https://github.com/atlas0fd00m/rfcat>.
- [3] Eric Blossom. "GNU Radio: Tools for Exploring the Radio Frequency Spectrum." In: *Linux journal* 122 (2004), p. 4.
- [4] Luca Bongiorni. *WHID Elite*. 2019. URL: <https://whid.ninja>.
- [5] Cesar Cerrudo, Esteban Martinez Fayo, and Matias Sequeira. *Do You Blindly Trust LoRaWAN Networks for IoT?* Feb. 13, 2020. URL: <https://ioactive.com/do-you-blindly-trust-lorawan-networks-for-iot/>.

- [6] Samy Kamkar. “Drive it like you hacked it: New attacks and tools to wirelessly steal cars.” 2015. URL: <https://samy.pl/defcon2015/2015-defcon.pdf>.
- [7] Federico Maggi et al. “A Security Evaluation of Industrial Radio Remote Controllers.” In: *Proceedings of the 16th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Ed. by Roberto Perdisci and Magnus Almgren. Vol. 11543. Gothenburg, Sweden: Springer International Publishing, June 19, 2019. ISBN: 978-3-030-22037-2. DOI: [10.1007/978-3-030-22038-9_7](https://doi.org/10.1007/978-3-030-22038-9_7).
- [8] Bastille Networks. “Firmware and research tools for Nordic Semiconductor nRF24LU1+ based USB dongles and breakout boards.” URL: <https://github.com/BastilleResearch/nrf-research-firmware>.
- [9] Mark Newlin. “Injecting keystrokes into wireless mice.” 2016. DOI: <https://doi.org/10.5446/36268#t=00:09,01:30>. URL: <https://av.tib.eu/media/36268>.
- [10] Michael Ossmann and Great Scott Gadgets. *Rapid radio reversing*. Tech. rep. Tech. rep, 2016. URL: <https://pdfs.semanticscholar.org/cc6d/fe7689e8eb1e1804776d111f9659818076b5.pdf>.
- [11] Johannes Pohl and Andreas Noack. “Universal Radio Hacker: A Suite for Analyzing and Attacking Stateful Wireless Protocols.” In: *Proceedings of the 12th USENIX Conference on Offensive Technologies*. WOOT’18. Baltimore, MD, USA: USENIX Association, 2018, p. 6. URL: <https://www.usenix.org/system/files/conference/woot18/woot18-paper-pohl.pdf>.
- [12] ComThings SAS. *PandwaRF*. URL: <https://pandwarf.com/>.

A Screenshots

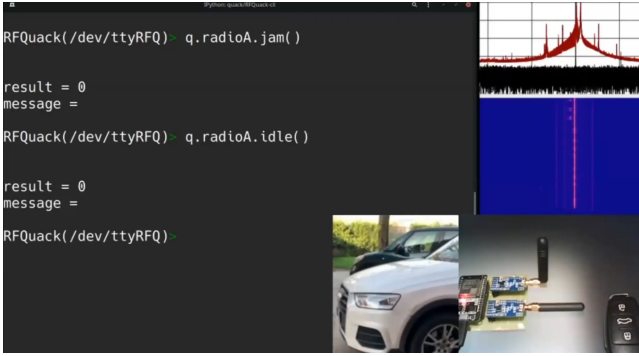


Figure 6: RollJam implementation on top of RFQuack in action, taken from durin presentation at CanSecWest.

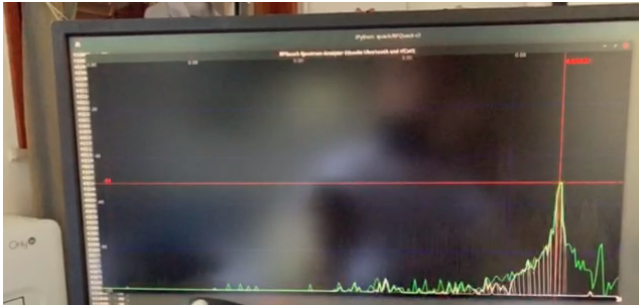


Figure 7: Spectrum slicing to find peaks and detect transmitting frequency, as described in Section 4.3.

B Transceiver-agnostic, Uniform API

RFQuack's support for multiple radios allows to use the functionalities of many different transceivers using the same API, which exposes the following groups of functions:

Radio Mode. Put the transceiver in standby, receiving, transmit, promiscuous, or jamming mode. In promiscuous mode, the radio will receive anything with RSSI above a user-configurable threshold (to avoid receiving pure noise), regardless of the preamble or sync-word values. The jamming mode just keeps transmitting, optionally leveraging transceiver-specific features to avoid busy loops (e.g., the CC11xx will keep transmitting as long as its TX FIFO is not empty).

RF Parameters. Set the carrier frequency, frequency deviation (for FSK modulations), RX bandwidth, and bitrate.

Packet Format. Set the fixed or variable packet length format, preamble length, sync-word value, and CRC check.

Carrier. Set the output power, get the last RSSI, determine if there is carrier at the tuned frequency.

TX and RX:. transmit a byte array or receive onto a byte array or queue. Optionally, activate RX loop and enqueue any received packet.

Register Access. Read or write values from or to the transceiver registers.

C Serialization and Transport Protocol

Protobuf is Google's cross-platform serialization protocol, which support most of the mainstream languages such as Python, Go, C, C++, Java. It allows to create truly interoperable protocols and makes it easy to interface systems across very different languages and architectures.

To show an example of the type system of RFQuack, we describe the `PacketModification` type, which is the core of RFQuack's packet-matching and modification engine.

```
// Tell the tool how to modify a byte of a packet.
message PacketModification {
  // position in the packet
  optional uint32 position = 1;

  // pos = all indexOf(content)
  optional uint32 content = 2;

  enum Op {
    // pkt[pos] = pkt[pos] & operand
    AND = 1;

    // pkt[pos] = pkt[pos] | operand
    OR = 2;

    // pkt[pos] = pkt[pos] ^ operand
    XOR = 3;

    // pkt[pos] = ~pkt[pos]
    NOT = 4;

    // pkt[pos] = pkt[pos] << operand
    SLEFT = 5;

    // pkt[pos] = pkt[pos] >> operand
    SRIGHT = 6;

    // pkt = payload + pkt
    PREPEND = 7;

    // pkt = pkt + payload
    APPEND = 8;

    // pkt = pkt[0 : pos] + payload + pkt[pos:pkt.size]
    INSERT = 9;
  }

  optional Op operation = 3;
  optional uint32 operand = 4;

  // Apply only to packets matching a pattern
  optional string pattern = 5;

  // Bytes to append / prepend for OP = PREPEND | APPEND
  optional bytes payload = 6;
}
```

The default RFQuack IPython shell has automatic auto-completion based on introspection of the Protobuf types. This means that new Protobuf types are automatically "added" and supported by the console, without changing any code.

To achieve RPC functionalities, the Protobuf messages are prepended by a URI that identifies the command (similarly to MQTT topics) exchanged between the host and the dongle. The resulting binary messages are transported over a serial connection (USB) and over MQTT for WiFi (and cellular).

D Console Scripting

In this section we provide some examples of RFQuack's IPython shell scripting capabilities, which we used in the case studies and experiments described in Section 5 and 6.

D.1 Frequency Recovery Experiment

This script, mentioned in Section 6.2, tunes both radios at 432–437 Mhz, transmits with the first radio and collects any packets received by the second radio, along with the detected carrier frequency.

```
q.radioA.rx() # set first radio in RX mode
q.guessing.start_freq = 432
q.guessing.end_freq = 437
q.guessing.start()
values = dict()
for i in range(4320, 4373, 3):
    freq = i/10
    values[freq] = list()
    q.radioB.set_modem_config(carrierFreq=freq)
    q.radioB.tx() # second radio in TX mode
    time.sleep(1)
    for times in range(0, 50):
        q.data = [] # Clear recv packets buffer
        q.radioB.send(data=bytes.fromhex("aaa[...].666"))
        time.sleep(0.5) # Wait for decoding
        values[freq].append(q.data[0].carrierFreq)
```

D.2 Bitrate Recovery Experiment

This script, mentioned in Section 6.3, sets one radio in RX mode and enables the bitrate recovery loop. Then it transmits the same packet at varying bitrate values, for 50 times each, letting the first radio clamp on the signal and estimate the bitrate.

```
q.radioA.rx() # set first radio in RX mode
q.guessing.sampling_bitrate = 30 #Then, 60
q.guessing.start()
values = dict()

for i in range(10, 150, 3):
    br = i/10
    values[br] = list()
    q.radioB.set_modem_config(bitRate=br)
    q.radioB.tx()
    q.sleep(1)
    for times in range(0, 50):
        q.data = [] # Clear received packets
        q.radioB.send(data=bytes.fromhex("aaa[...].666"))
        time.sleep(0.5) # Wait for decoding
        # Decoded packets get stored in q.data
        values[br].append(q.data[0].bitRate)
```

D.3 Isolate 2.4GHz Valid Frames

This script set the nRF24 2.4GHz frontend in promiscuous mode at 2000 kbps, collects all the sync words and count their occurrences. Using the most frequent sync words, we queried the FCC database and found a lead that helped us isolate the traffic of the target device.

```
q.radioA.set_modem_config(bitRate=2000, isPromiscuous=True)
```

```
q.radioA.set_packet_len(isFixedPacketLen=True, packetLen=32)
# max
q.radioA.rx()

# capture anything within the range
for freq in range(2405, 2474):
    q.radioA.set_modem_config(carrierFreq=freq)

# isolate all sync word and rank
sw = list(map(lambda x: x.data.hex()[0:10], q.data))
counter = Counter(sw)
```