RSSISPY: Inspecting Concurrent Transmissions in the Wild

Carsten Herrmann
Networked Embedded Systems Lab
TU Dresden, Germany
carsten.herrmann@tu-dresden.de

Marco Zimmerling
Networked Embedded Systems Lab
University of Freiburg, Germany
zimmerling@cs.uni-freiburg.de

Abstract

This paper presents RSSISPY, a software module that enables continuous RSSI measurements with very high time resolution on standard low-power wireless network nodes. Specifically, RSSISPY can be used to perform continuous RSSI sampling in parallel to normal Bluetooth Low Energy (BLE) receive operation on Nordic nRF52840 devices with a sampling rate of 1 Msps, i.e., one sample per bit in BLE 1M mode. RSSISPY provides this functionality without additional hardware and can be straightforwardly integrated into existing protocols, so it allows to record RSSI traces "in the wild", i.e., in existing networks running the protocols of interest. We use RSSISPY to further investigate important physical layer effects on COTS hardware, including interference in general, capture effect, and so-called constructive interference. Our systematic experiments combined with rigorous analyses uncover results that have not been known so far and are of high interest for concurrent-transmissions-based protocols. Besides that, RSSISPY enables all kinds of RSSI-stream-based functionality on standard hardware, e.g., signal detection, parameter estimation (instantaneous SNR, transmitter constellation, etc.), and cross-technology communication.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*wireless communication*; C.4 [**Performance of Systems**]: *measurement techniques*

General Terms

Design, Experimentation, Measurement

Keywords

Bluetooth (BLE), RSSI Measurement, Synchronous Transmissions, Capture Effect, Cross-Technology Communication

1 Introduction

In an era of automation and digitalization in critical domains (manufacturing, medicine, etc.), ensuring the dependability of low-power wireless communication solutions is a key concern. In response, the EWSN Dependability Competition provided a forum for teams from academia and industry to compare the dependability of their low-power wireless protocols in the presence of significant environment dynamics [9]. Solutions based on *synchronous transmissions (ST)* routinely took up the first three places in the competition. Meanwhile, ST have also been exploited to provide formal functional guarantees that were previously only known from wired communication systems [24].

Despite over a decade of research, however, the foundation of these protocols—the concept of ST—is *still* not well understood. Regarding the selected physical layer, what is it that enables a commodity low-power wireless radio to receive (or not receive) a packet despite the interference caused by ST? What is the impact of the modulation scheme, the senders' and receiver's hardware implementation, and the properties of the wireless channel? While progress on these and other questions has been made, conclusive answers backed up by a combination of rigorous analyses and systematic experiments are still being sought. Without such answers, the societal acceptance of low-power wireless technology for mission- and safety-critical applications is at stake.

As discussed in Sec. 2, there is a significant gap between theoretical results on one hand and practical experiments on the other. It turns out that theoretical models and simulations tend to miss important aspects of real technology, particularly of the commercial off-the-shelf (COTS) devices used in real applications. Consequently, there is a substantial need for validating experiments on the real target, i.e., using the exact same radio hardware (and antenna position) to "look through the device's eyes". However, COTS devices come along with limited information about architectural details and missing access to internal signals, which makes such experiments difficult and hinders the interpretation of observed results. What is missing is a method to *deeply* inspect synchronous transmissions *"in the wild"*, i.e., in the target environment, on the deployed COTS devices, without relying on additional hardware, and (ideally) executing the application of interest.

Contribution and roadmap. To reduce this gap, we present RSSISPY, a software module that allows to gain insights into ST that have not been possible before. RSSISPY, running on standard nRF52840 devices in BLE mode, adds continuous *receive signal strength indicator (RSSI)* sampling to existing packet reception routines and—as the first solution of this

kind—provides bit-level time resolution by supporting sample rates up to 1 Msps. Thus, incorporating RSSISPY into existing protocols not only allows to perform joined bit error and RSSI analysis "in the wild", it also provides a level of detail on commodity devices that has been unseen so far.

We present the details of RSSISPY in Sec. 3 and use it in Sec. 4 to investigate ST. The power of RSSISPY allows us to dissect the nature of physical-layer effects in an unprecedented way, which provides new insights and inspires new ideas. For instance, regarding the *capture effect* we not only find that the *capture window* is longer than expected, but that it can even be extended to an arbitrary length.

For each experiment, we deduce our expectations based on rigorous analytical considerations and discuss the relation to our empirical results. One important issue is the beating pattern that results from the *carrier frequency offsets (CFOs)* of multiple synchronous transmitters. In Sec. 4.5.1 we analytically derive the shape of this pattern for the general case of arbitrary many transmitters. This contribution generalizes existing analytical results, which focus only on the special case of two transmitters [17, 7]. We discuss other aspects of RSSISPY in Sec. 5 and conclude in Sec. 6. In summary, this paper contributes

- RSSISPY, a software tool[1] that enables new ways to investigate CT in the target environment by providing RSSI sampling with bit-level time resolution on COTS devices
- new results on physical layer effects that are of high interest for CT-based protocols.

2 Background and Related Work

At its core, RSSISPY is a tool for continuous high-resolution RSSI sampling on low-power COTS devices. Continuous RSSI sampling (CRS) is beyond the scope of typical protocol stacks, where RSSI measurements are performed only once per packet (if at all). However, it is popular in *cross-technology communication (CTC)*, as it is one of the enablers in this field [13]. For example, CRS-based CTC has been implemented on low-power devices in [10, 15, 14, 12]. Unfortunately, none of these solutions reaches sample rates above 50 kHz, while RSSISPY provides 1 MHz—a 20 times improvement over the state of the art.

In this paper, we use RSSISPY for an in-depth inspection of *concurrent transmissions (CT)* in BLE networks. CT is a communication technique that lets multiple nodes transmit packets at the same time, i.e., overlapping or even simultaneously (symbol-synchronous). The latter is referred to as *synchronous transmissions*. In the last ten years it has been shown that CT is particularly efficient in low-power wireless multi-hop networks, and the pioneering work of Glossy [11] triggered the exploitation of CT in a variety of protocol designs (an extensive survey can be found in [24]).

The benefit of CT has been attributed to the *capture effect* and, in case of ST, to *constructive interference (CI)* (we discuss the details in Sec. 4). Both effects have been investigated in practical experiments [23, 17, 18, 8] as well as in simulations [21, 16, 8] and—to a limited extent—analytically [21, 22, 7]. However, overlooking or omitting important details on both sides has lead to inconsistent interpretations regarding ST, and over time a controversy on the "full truth" arose [22, 17, 16, 18]. Today it is known that the unavoidable CFO between multiple transmitters is one of the critical factors [17, 16, 8, 7], which went unnoticed in the early days. Probably there are more crucial details that are unknown so far (e.g., regarding the precise modulation schemes and demodulator architectures).

RSSISPY is a software tool that grants access to the continuous RSSI signal on COTS devices, which is most interesting for the investigation of CFO effects and CT in general and has been out of reach so far. The value of RSSISPY is highlighted by the unprecedented results in Sec. 4, which, hopefully, inspire and push further research in this area. Besides that, we believe that practical and theoretical work must be consolidated in a more stringent way. To this end, we align all our experiments with a consistent mathematical derivation. One advantage of this combined approach is demonstrated in Sec. 4.5.3, where it prevents a misinterpretation of our experimental results.

3 RSSISPY in Detail

In this section we present the internal details of RSSISPY. We first introduce nRF52840's hardware support for reading single RSSI samples and discuss how it could be combined with other peripheral modules to implement continuous sampling (Sec. 3.1). We then consider the timing of this concept and analyze all uncertainty factors that must be taken into account for a reliable operation. This requires an in-depth discussion of the relevant hardware details (Sec. 3.2) and reveals that establishing a 1 MHz sample rate is remarkably challenging (Sec. 3.3). We unfold the internal design of RSSISPY in Sec. 3.4 and explain how it overcomes all challenges.

3.1 Basic Operation and Timing

The nRF52840 radio peripheral implements a simple mechanism for measuring RSSI values [6]. Internally, the RSSI signal is measured continuously and passed through a low-pass filter. The output value of this filter can be sampled on demand by triggering the RSSISTART task. After a sampling period T_S the RSSIEND event is generated and the sampled value can be read from the RSSISAMPLE register.

Continuous sampling can be realized by connecting a timer peripheral's COMPARE event to the RSSISTART task via nRF52840's *Programmable Peripheral Interconnect (PPI)* [6]. Essentially, the PPI is a configurable interconnect matrix that allows to route event signals between different peripheral units, so events of one peripheral can trigger tasks of other peripherals without CPU interruption. In spite of that, the CPU is needed to read each RSSI sample from the RSSISAMPLE register and write it to a circular buffer. To this end, an obvious solution is to generate a radio interrupt on each RSSIEVENT and move the data within the corresponding *interrupt service routine (ISR)*. The CPU of the nRF52840 consists of an ARM Cortex-M4F core running with a frequency of 64 MHz. Targeting an RSSI sampling rate of 1 MHz there are 64 CPU clock cycles to generate and process each sample.

Figure 1 illustrates this straightforward implementation. Unlike the CPU, the peripheral subsystem of the nRF52840 is driven by a 16 MHz clock (PCLK16M). Timer 1 is used as sample clock and generates a COMPARE event every 1 μs.

[1] The source code is available at `https://gitlab.com/nes-lab/rssispy`.

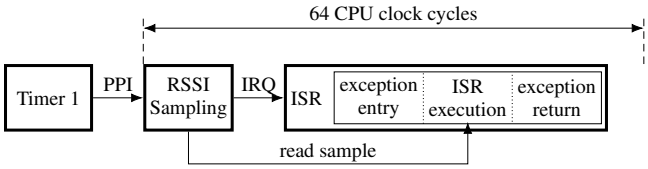

Figure 1: Naive implementation of continuous RSSI sampling on the nRF52840.

The event is routed to the RSSISTART task via the PPI, which introduces a fixed delay of 1 PCLK16M = 4 CPU clock cycles. The sampling period T_S is specified as typically 0.25 μs = 16 CPU clock cycles in [6]. However, in all our experiments the RSSIEND event appeared with a constant delay of 4 cycles (1 PCLK16M cycle), so we assume this is the right value. The ISR execution time can be divided into the exception entry delay (aka interrupt latency), the execution time of the ISR itself, and the exception return delay.

Among others, the ISR contains a load instruction that reads the RSSISAMPLE register. *Ensuring that this instruction is always executed at the right time*, i.e., after sampling of the current value has finished and before the next sampling cycle gets started, is key for a reliable operation. What sounds not that difficult in the first place reveals itself as a significant challenge when looking closer. To understand this, we must dive into the exception handling and timing details of the ARM Cortex-M4F core as well as the peripheral register access timing of the nRF52840.

3.2 nRF52840 Architecture Details

3.2.1 ARMv7-M Exception Handling

The Cortex-M4F processor core [2, 4] implements the microcontroller profile of ARM's version 7 architecture (ARMv7-M) [5], which contains a *Nested Vectored Interrupt Controller (NVIC)*. The NVIC is closely coupled with the core's exception handling, and in combination both provide some special features that are very important for RSSISPY.

As a starting ground, the NVIC comes with programmable interrupt priority levels and hardware support for context save and restore. When there is a pending *interrupt request (IRQ)* with sufficient priority (for the moment let us assume that there is no other IRQ), then the program flow gets interrupted and exception handling begins. The processor pushes context information (machine state, program counter, etc.) onto the stack, which is referred to as *stacking*. In parallel (ideally), the processor reads the ISR's start address from the exception vector table and fetches the first instructions to feed the CPU pipeline. After stacking has finished, the processor starts executing the ISR. At the end of the ISR the processor initiates an exception return sequence. It restores the saved context from the stack and, in parallel, fetches the next instructions of the interrupted program.

Under ideal conditions there is a maximum of a 12 cycle latency from asserting the IRQ to execution of the first ISR instruction and a 10 cycle latency for the return sequence [2]. This is reachable if (i) the memory subsystem provides optimal performance, i.e., congestion-free full parallel instruction and data busses without any interlocks or wait states, and (ii) the processor does not stack additional context information for a *floating point unit (FPU)*. The Cortex-M4*F* has an FPU, so (ii) must be taken into account. Otherwise, the latencies increase to 29 and 27 cycles for entry and return, respectively.

When a higher priority IRQ arrives while an exception is being handled, then the new exception preempts the original exception. The processor's advanced features make it necessary to distinguish three cases here. (i) If the IRQ is asserted while the ISR is running, i.e., between the entry and return sequence of the original exception, then the latencies for the new exception are as discussed above. (ii) If the IRQ is asserted during the entry sequence of the first exception, then *late-arrival* handling reduces the entry latency of the new exception by a variable number of cycles.[2] (iii) If the IRQ is asserted during the return sequence of the first exception, then *tail-chaining* reduces the entry latency of the new exception to 6 cycles. We omit a detailed discussion of these advanced features, the interested reader is refered to [4, 5].[3] An important consequence for RSSISPY is that we have to cope with significantly fluctuating entry and exit latencies.

3.2.2 nRF52840 Peripheral Access Timing

The connection between the Cortex-M4 core and the peripheral units is organized memory-mapped, i.e., the CPU uses standard load and store instructions to read and write peripheral registers. In the nRF52840 such memory transfers pass an AHB multilayer interconnect and an AHB2APB bus bridge [6]. While the CPU core runs with 64 MHz, the Advanced Peripheral Bus (APB) is clocked by PCLK16M, which slows down accesses significantly. The nRF52840 manual [6] does not provide much information about this topic, so we implemented a number of micro benchmarks to determine the peripheral load/store timing. Our findings are as follows.

- Each peripheral access is synchronized to PCLK16M. Hence, depending on the current phase relation between the instruction stream and PCLK16M, there is a synchronization delay of 0 . . . 3 CPU clock cycles.
- Peripheral accesses take 2 PCLK16M cycles, which translates into 8 wait states at the CPU side.
- Peripheral loads are not pipelined with other loads (different from normal loads).
- Peripheral stores use the CPU core's one-entry write buffer (as normal stores do).
- Peripheral accesses are not interruptable.

In consequence, a load from a peripheral register takes between 10 and 13 CPU clock cycles, while a store blocks consecutive accesses for 11 to 14 cycles. Further, peripheral accesses can increase the interrupt latency by up to 13 cycles.

3.3 Challenges

In the face of the hardware details discussed above, it is easy to understand that regarding the design of RSSISPY

[2] The exact minimum latency is unclear [1]. Due to the vector fetch it must be at least 2. We guess it is 6, the same as with tail-chaining.

[3] Both mechanisms exploit the fact that the context to be saved is independent of the taken exception, only the vector table fetch must be adopted. Hence, the outcome of a stacking operation can be used to take a different (late-arriving) exception than originally planed, and a stacked context can be reused to take another exception without initiating a new stacking operation.

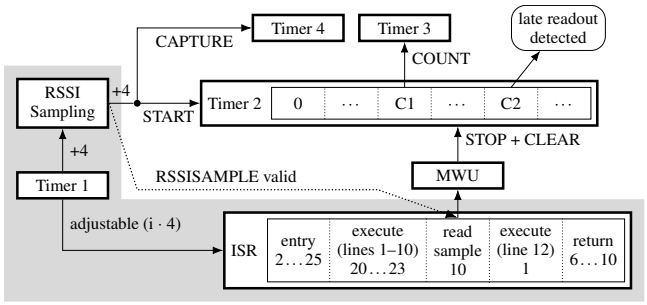

Figure 2: RSSISPY's architecture. *The gray part realizes RSSI sampling, the other blocks form the monitoring system. The numbers in the ISR denote execution times (CPU cycles).*

almost everything is about timing. To reliably record a RSSI sample every 64 CPU clock cycles,

(C1) The exception handling and memory subsystems must be used most efficiently, i.e., without any bus congestion or FPU context stacking.

(C2) Apart from inevitable wait states, the ISR must perform all operations in only a few CPU clock cycles. Exception entry, return, IRQ acknowledgment (a peripheral store), and RSSISAMPLE reading (a peripheral load) can consume up to 62 of the 64 clock cycles, so the remaining operations—which include address generation and circular buffer handling—must be extremely fast.

(C3) In addition to (C2), the implementation must ensure that fluctuating interrupt latencies caused by tail-chaining and late-arrival handling cannot shift the RSSISAMPLE read operation out of the tolerable time frame.

Besides those strictly timing-related challenges, there are also some functional aspects.

(C4) Since the circular buffer can overflow, the implementation must provide a timestamp that indicates the temporal placement of the buffered samples.

(C5) We support the integration of RSSISPY into other projects, and human integrators can make mistakes (e.g., set wrong interrupt priorities). Therefore, the implementation should be able to detect invalid or missed samples.

(C4) and (C5) would be easy if the ISR could generate timestamps for each sample, but this is impossible due to (C2), as reading a timer value corresponds to another peripheral load and takes way too long. Hence, the functionality must be established outside of the CPU core, which makes it tricky.

3.4 Implementation

In the following we focus on RSSISPY's core, wich consists of an RSSI sampling ISR surrounded by an arrangement of peripheral units, the latter being connected via several PPI channels (Fig. 2). This core is complemented by appropriate data structures, initialization routines, etc., which we skip here. We also omit details regarding graceful start and stop mechanisms, temperature compensation, and pre- and post-trigger functionality used in our experiments (Sec. 4). Before we discuss Fig. 2 in detail, we address (C1).

Listing 1: RSSISPY's sampling ISR

```
1   mov.w     r0,  #0x40000000
2   add.w     r3,  r0, #<TIMER>
3   movw.w    r2,  #<BUFFER[0:15]>
4   str.w     r0,  [r3, #<TIMER_EVENT>]
5   movt.w    r2,  #<BUFFER[16:31]>
6   add.w     r0,  r0, #<RADIO>
7   ldr.w     r3,  [r2, #<NUM_WRITTEN_OFFSET>]
8   add.w     r12, r3, #1
9   str.w     r12, [r2, #<NUM_WRITTEN_OFFSET>]
10  bfc.w     r3,  #<BUF_SIZE>, #<32 - BUF_SIZE>
11  ldr       r0,  [r0, #<RSSISAMPLE>]
12  strb r0,  [r2, r3, lsl #0]
13  bx        lr
```

3.4.1 Optimized Memory Access and Lazy Stacking

In a typical nRF52840 application the machine code runs directly from flash memory. However, executing code from flash comes with a wait state penalty [6], so this is not an option for the RSSISPY ISR, which must accordingly run from RAM. This, in turn, has a pitfall, because the shortest interrupt latency can only be achieved if the vector and instruction fetches can be parallelized with stacking. To solve this, we make use of nRF52840's multilayer interconnect and split an 8 kB section from the data RAM to have it available for fast instruction fetches. We omit the details here, the split is realized via the linker script and also includes address range overlays to assign critical accesses to different busses and ensure that they get parallelized. We also relocate the exception vector table to this RAM section to ensure maximum vector fetch performance.

Straightforwardly extending the exception handling discussed in Sec. 3.2.1 to FPU-equipped processors would require to save the FPU context with each exception. The Cortex-M4F implements special (optional) features that aim at circumventing this expensive step, among others so-called *Lazy Stacking*. Due to space constraints we cannot go into the details here, the interested reader is referred to [5, 3]. In essence, lazy stacking allows to defer stacking until an FPU instruction is executed. We enable this feature and exploit it by combining two things: RSSISPY's IRQ is the highest priority IRQ in the system, and the ISR is written in pure assembly code (Listing 1), so it is known for sure that it does not use FPU instructions. Consequently, stacking will never occur while the exception is active, which effectively avoids extended entry and return latencies.

3.4.2 Fast RSSI Sampling

Timer 1 generates the sample clock. It implements a 4 bit counter and generates a COMPARE event at every wrap-around, which is connected to the RSSISTART task via the PPI. In favor of RSSIEND (as in Sec. 3.1), we use a second compare register of Timer 1 to trigger the sampling IRQ. This enables us to shift the IRQ relative to RSSIEND (i.e., to pre- or post-trigger the ISR) and thereby—targeting (C3)—to place the RSSISAMPLE read instruction in a safe time frame.

The sampling ISR is shown in Listing 1. To gain the maximum performance, three instructions (lines 3, 5, and 10) are updated at runtime to adopt the intended destination buffer (once before sampling is started), which is possible

because the ISR is placed in RAM (Sec. 3.4.1). The ISR works as follows. First, it computes needed pointers (lines 1–6) and acknowledges the IRQ (line 4). The peripheral store is expensive, however, due to the core's write buffer it gets decoupled and the execution up to line 7 continues in parallel. Register `r2` refers to a buffer structure of the form

```
struct Rssi_Buffer {
     ...              // header data
     uint32_t  num_samples_written;
     ...              // more header data
     uint8_t          samples[];
};
```

and contains a pointer to the `samples` member. The instruction in line 7 loads the field `num_samples_written` from the structure, where `<NUM_WRITTEN_OFFSET>` equals to `offsetof(Rssi_Buffer, num_samples_written) - offsetof(Rssi_Buffer, samples)`. The field is incremented and written back (lines 8–9), before its original value is masked with `sizeof(samples) - 1` in line 10 (setting `<BUF_SIZE> = MSB(sizeof(samples))`). The latter corresponds to a modulo operation if `sizeof(samples)` is a power of 2 and realizes a very fast circular buffer indexing mechanism including wrap-around handling. The result is used as a register offset in line 12 to store the RSSI sample loaded in line 11 to the right buffer position. Line 13 initiates the exception return sequence. We want to emphasize the circular buffer handling concept, which uses `num_written` (`nw`) and `num_read` (`nr`) counters instead of classic read and write pointers. Besides its efficiency, this allows to implement the writer (i.e., the ISR) without overflow handling, as the latter can be done on the reader side as follows (`bufsize = sizeof(samples)`).

```
nr = (nw < bufsize) ? 0 : nw - bufsize;
// printf("%u samples lost\n", nr);
while (nr != nw)
     data = samples[nr++ & (bufsize - 1)];
```

Based on the ISR assembly code and considering all related details (including the core's instruction pipeline characteristics), we determined cycle-accurate execution times of all ISR parts. The results are illustrated in Fig. 2. As a side effect, our in-depth analysis enabled us to arrange the machine instructions such that the peripheral load in line 10 gets executed as fast as possible (compare Sec. 3.2.2).[4] With normal entry and exit latencies the read operation starts 32–35 cycles after the IRQ, and overall the exception handling takes 53–56 cycles. In the worst-case the exception handling seems to take 69 cycles, i.e., it seemingly exceeds the available 64 cycles. In such cases we exploit the processor's advanced exception handling features. The *next* IRQ, which then overlaps the return phase of the *first* exception, triggers late-arrival handling. Consequently, the context unstacking is aborted, re-stacking is skipped, and the *second* ISR starts not later than 6 cycles after the *second* IRQ. So in sum it takes not more than $64+6+44 = 114 < 128$ cycles to handle *both* IRQs.

Overall, our sampling concept effectively overcomes (C2) and (C3), provided that the sampling IRQ is configured to have the highest priority, and that interrupts do never get globally disabled while sampling is active. The latter can be achieved easily by using the core register BASEPRI instead of PRIMASK to implement interrupt locks.

[4] The key is that the peripheral store in line 4, which has an uncertainty of 3 cycles, implicitly synchronizes the instruction stream with PCLK16M.

3.4.3 Zero-Load ISR Monitoring

As discussed in Sec. 3.3 and underlined by Sec. 3.4.2, there is no chance to integrate timestamping or monitoring functionality into the ISR. Hence, we use additional peripheral units for this purpose, as shown in Fig. 2. We first focus on (C5), as (C4) can be handled easily afterwards.

Marking individual samples as invalid or missing would require ISR support, so this is impossible. However, such samples should not occur in a properly configured system, and their appearance can be rated as the exceptional case. Therefore, it is enough to detect if *any* sample in a recorded block is invalid or missing (declaring the block as not trustworthy).

To this end, we make use of the nRF52840's *Memory Watch Unit (MWU)*. The MWU is a special peripheral tightly coupled to the CPU that can be used to generate peripheral events whenever the CPU accesses specified memory regions. We exploit this capability to monitor the timing of the load instruction in line 11 (Listing 1) in each ISR execution. Specifically, the load must take place when the internal count register of Timer 2 is inside the half-open interval delimited by the compare registers C1 and C2 (Fig. 2). We call it a *late readout* if the timer value passes C2. A late readout in a recording can be detected easily by checking the event flag that belongs to C2 after recording has been stopped. On the other hand, not passing C1 is referred to as an *early readout*, which can indicate an internal misconfiguration (Timer 1 triggers the IRQ too early). Detecting early readouts is more tricky, as the event flag does not help here. At the beginning of the measurement we capture the time when Timer 1 gets started, t_0, in the main clock Timer 4 (not shown in Fig. 2). Further, we capture the timestamp of each RSSIEND event in Timer 4, so after stopping this value provides the timestamp of the last sample, t_N. The difference between t_0 and t_N indicates the expected number of samples N, i.e., the number of samples that should have been recorded. Now, we route the COMPARE event attached to C1 to Timer 3. This timer runs in counter mode and counts how often Timer 2 passed C1. If Timer 3 does not equal N, then there have been early readouts.

As a side effect, after recording has been stopped the difference between N and `num_samples_written` (Sec. 3.4.2) indicates the number of missed samples. Further, if there have been neither early nor late readouts we can reliably compute the timestamp of each recorded sample backwards from t_N, which effectively overcomes (C4).

4 RssiSpy in Action

To demonstrate the utility of RssiSpy, we use it to investigate physical layer effects that appear in the presence of CT. In particular, besides the influence of interference in general, we elaborate on the capture effect and so-called constructive interference. Our goal is to dissect the inner nature of these effects. To this end, we conduct controlled experiments that systematically provoke critical situations and use RssiSpy to add high-rate RSSI measurements to the recorded packets. This enables us to present results with a level of detail that has been unseen so far on COTS hardware, which is of high interest for CT-based protocols. Before we dive into the ex-

periments, we derive the fundamental mathematical relations to be able to deduce our expectations before each experiment and compare them with the empirical results.

4.1 A Primer on Radio Channel Modeling

In general, a modulated radio transmit signal can be expressed as [19]

$$s(t) = a(t)\cos(2\pi f_{c,T} t + \phi(t)) \tag{1}$$

Here, $f_{c,T}$ is the transmitter's carrier frequency, and a and ϕ denote amplitude and phase modulation functions that reproduce the used modulation scheme. For example, in amplitude shift keying $\phi(t)$ is constant and $a(t)$ equals the amplitude that represents the symbol transmitted at time t. In simple frequency shift keying (FSK) variants $a(t)$ is constant and $\phi(t) = 2\pi f_i t$, where f_i is the frequency representing symbol i.

To ease analysis, it is common to interpret $s(t)$ as the real part of a complex signal, more specifically of a complex carrier $e^{\mathrm{j}2\pi f_{c,T} t}$ modulated by a complex envelope $u(t) = a(t)e^{\mathrm{j}\phi(t)}$, i.e., $s(t) = \mathrm{Re}\left(u(t)\, e^{\mathrm{j}2\pi f_{c,T} t}\right)$. Further, denoting the real and imaginary parts of $u(t)$ by $I(t)$ and $Q(t)$, respectively, $s(t)$ can be rewritten as [19]

$$s(t) = I(t)\cos(2\pi f_{c,T} t) - Q(t)\sin(2\pi f_{c,T} t) \tag{2}$$

which represents the functionality of an IQ upconverter found in many modern radio implementations.[5] Its counterpart at the receiver, the downconverter, performs the similar operation

$$v(t) = r(t)\cos(2\pi f_{c,R} t) - \mathrm{j}\, r(t)\sin(2\pi f_{c,R} t) \tag{3}$$

to extract the complex envelope v from the receive signal r.

In the simplest model, r is an attenuated and time-shifted version of s, i.e.,

$$r(t) = |h|\, s(t-\tau) \tag{4}$$

with $|h| < 1$ denoting the gain (attenuation) of the link and τ the overall delay due to time-of-flight, transmitter latency, etc.. It is straightforward to show that under this model, with v passing an appropriate low-pass filter, (3) can be rewritten as

$$\begin{aligned} v(t) &= 0.5\,|h|\, u(t-\tau)\, e^{\mathrm{j}(2\pi(f_{c,T}-f_{c,R})t + \phi_{0,T} - \phi_{0,R})} \\ &= 0.5\,|h|\, u(t-\tau)\, e^{\mathrm{j}\Delta\phi_0} e^{\mathrm{j}2\pi\Delta f_c t} \end{aligned} \tag{5}$$

where $\Delta f_c = f_{c,T} - f_{c,R}$ denotes the carrier frequency offset (CFO) between sender and receiver, and $\Delta\phi_0 = \phi_{0,T} - \phi_{0,R}$ denotes the phase difference. $\phi_{0,T}$ includes the initial phase of the transmitter's oscillator, the delay-induced portion $e^{-\mathrm{j}2\pi f_c \tau}$, and potential phase shifts resulting from the wireless channel (e.g., caused by reflections). Since we are only interested in the resulting phase difference relative to the receiver, we can neglect those details and merge all time-independent parts into a complex channel coefficient h, leading to

$$v(t) = h\, u(t-\tau)\, e^{\mathrm{j}2\pi\Delta f_c t} \tag{6}$$

[5] Though, this analysis does not require that the transmitter and receiver use IQ conversion, the signals s and r can be generated and processed in different technical ways. Instead, the analysis is meaningful because eventually all realizations lead to signals as in (1), and the wireless channel is independent of the used (de-)modulator technology.

The goal of a receiver is the demodulation of u (i.e., the reconstruction of the information encoded therein). As an intermediate step, a straightforward receiver tries to reconstruct $u(t)$ from (6) by (i) estimating and compensating for $|h|$, which is known as *Automatic Gain Control (AGC)*, (ii) eliminating the exponential term by synchronizing $f_{c,R}$ with $f_{c,T}$ in order to approach $\Delta f_c = 0$, and (iii) estimating and compensating for $\arg(h)$ and τ in a modulation-specific way.

When N nodes transmit concurrently, the received signal is a superposition of the transmitted signals with individual gains and delays for each path, i.e.,

$$r(t) = \sum_{k=1}^{N} |h_k| s_k(t-\tau_k) \tag{7}$$

We can derive the downconverted receive signal v by inserting (7) into (3) and rearranging the addends, which leads to

$$\begin{aligned} v(t) &= \sum_k \underbrace{|h_k|\left(s_k(t-\tau_k)\cos(2\pi f_{c,R} t) - \mathrm{j} s_k(t-\tau_k)\sin(2\pi f_{c,R} t)\right)}_{\overset{(3),(4)}{=} v_k(t)} \\ &\overset{(6)}{=} \sum_{k=1}^{N} h_k u_k(t-\tau_k) e^{\mathrm{j}2\pi\Delta f_{c,k} t} \end{aligned} \tag{8}$$

Obviously, v is a superposition of (6), i.e., a linear combination of individually attenuated and delayed transmit signals u_k, where each of them is mixed with a residual frequency $\Delta f_{c,k} = f_{c,k} - f_{c,R}$. The latter makes an important difference compared to the single-transmitter case $N = 1$ because the receiver—which is not aware of multiple transmitters—cannot eliminate different $\Delta f_{c,k}$ values with only a single parameter $f_{c,R}$. This holds even if all senders transmit the same signal $u_k = u$ with the same delay $\tau_k = \tau$, as ST-based protocols try to do. We will come back to this in Sec. 4.5.

4.2 Experimental Setup

We perform all experiments on FlockLab [20]. FlockLab is an open static testbed, which makes it easier to reproduce our results (though link qualities still fluctuate). The usage of such standard deployments is only possible because RSSISPY (i) does not rely on additional hardware, and (ii) targets a device that is widely used in established testbeds. Every experiment is performed with an appropriate subset of nodes that contains interesting link constellations for the particular experiment. Figure 3 provides an overview of the testbed and all used nodes. We use BLE 1M mode with center frequencies of 2401 MHz and 2483 MHz, which are close to the corners of the ISM band and contain little external interference.

For our experiments we embed RSSISPY into a test application that performs the required transmit and receive operations at each node following a fixed schedule. The latter makes it easy to provoke the situations of interest as well as to estimate the oscillator drift of each node and to synchronize their clocks based on packet reception timestamps. The application logs each transmitted and received packet together with the sampled RSSI data. We use this data to detect bit errors and generate plots that show the RSSI level alongside the received bitstream.

To illustrate the basic functionality, Fig. 4a shows the reception of a short BLE packet sent by node 2 and received

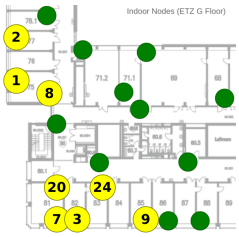

Figure 3: FlockLab floor plan (excerpt). *The yellow nodes are used in our experiments (the green nodes remain unused).*

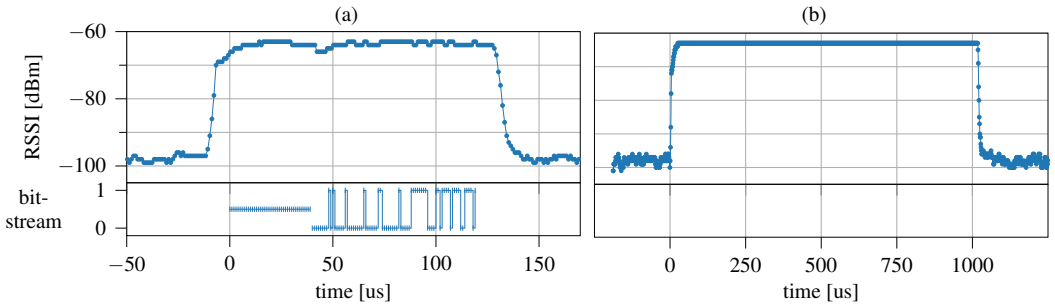

Figure 4: RSSISPY in action. *(a) Output of node 1 while receiving a packet from node 2 with parallel running* RSSISPY *($f_c = 2401$ MHz). The BLE link layer packet consists of 8 bit Preamble, 32 bit Access Address, 7 byte PDU, and 24 bit CRC. The preamble and address bits are shown with value 0.5 because we can only make assumptions about their real values.*[6] *The PDU bytes (`00 05 01 02 03 04 FF`) are transmitted LSB-first, the second PDU byte is the length field. (b)* RSSISPY *output at node 1 while node 2 transmits a pure unmodulated carrier signal. The signal does not form a valid packet or synchronization sequence (preamble + address), so the receiver outputs no bitstream.*

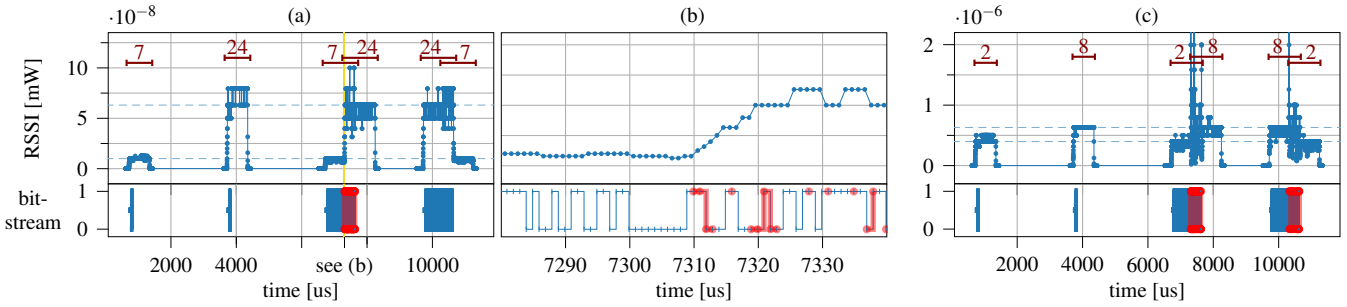

Figure 5: Destructively interfering transmissions. *Bit errors are highlighted in red.* $\overset{x}{\sqcup}$ *marks transmit activity at node x. (a) Transmissions of nodes 7 and 24 arriving at node 3 ($f_c = 2483$ MHz). The receive power of node 24 is about 7x (8.5 dB) higher compared to node 7. The weak signal does not disturb a running reception of the strong packet [@10000 µs], but if the strong signal appears in the middle of a weak packet [@7500 µs] it inhibits successful reception. (b) Zoom into plot (a) around 7310 µs. (c) Transmissions of nodes 2 and 8 arriving at node 1 ($f_c = 2401$ MHz). The receive power differs by a factor of only 1.5 (2 dB). Both signals disturb each other when received concurrently.*

at node 1. It is clearly visible how the RSSI increases from the noise level during the transmission. The transmit power was set to 0 dBm, so the link gain can be estimated as -63 dB. We also observe increased signal power shortly before and after the packet, which reveals the ramp-up and ramp-down times of the transmitter's radio. Note that those would not be visible without RSSISPY.

In addition to "classic" transmissions we allow transmitters to send pure, unmodulated carrier signals (before and after packets as well as stand-alone). This enables us to activate the transmit power for an arbitrary long time frame (not limited by the maximum packet length). A demonstration of this feature is shown in Fig. 4b. Amongst others, we use it in Sec. 4.5 to reveal important relations between packets and beating patterns that would be impossible to identify otherwise.

4.3 Reception with Arbitrary Interference

In a first experiment we consider interference in general. We explore how RSSISPY confirms the well known fact that small interference has almost no effect, but it gets destructive if the signal-to-interference ratio falls below a certain level.

Scenario. Node A starts a transmission. A listening node synchronizes to the transmitted signal and starts reception. In the middle of the packet some node B starts another transmission.

Expectation. The listening node synchronizes to node A, i.e., $f_{c,R} = f_{c,A} \Leftrightarrow \Delta f_{c,A} = 0$ and h_A and τ_A get compensated. Hence, (8) reads as $v(t) = u_A(t) + h_B/h_A \cdot u_B(t - (\tau_B - \tau_A))\, e^{j2\pi\Delta f_{c,B} t}$. If $|h_B| \ll |h_A|$, then the second addend is a tameable noise term, and the packet from A can be successfully received. However, if $|h_B|$ gets large, the arbitrary shape of the disturbance will inhibt successful reception.

Experiment. We perform the experiment with nodes 7 and 24 transmitting to node 3 and with nodes 2 and 8 transmitting to node 1 (Fig. 3). First, nodes A and B separately transmit a short packet (extended by a pure carrier period, see Sec. 4.2), which is used to estimate the receive power of both nodes. Then, both nodes transmit overlapping packets (time-shifted 600 µs to each other) with random payloads.

Results. The results are shown in Fig. 5 (note that we switched the vertical axis from dBm to mW, which stretches the granularity of high values). Node 24 is received at node 3 with considerably higher power than node 7. In consequence, an

[6] We know the desired values, but the radio does not provide a way to read the actually received bits.

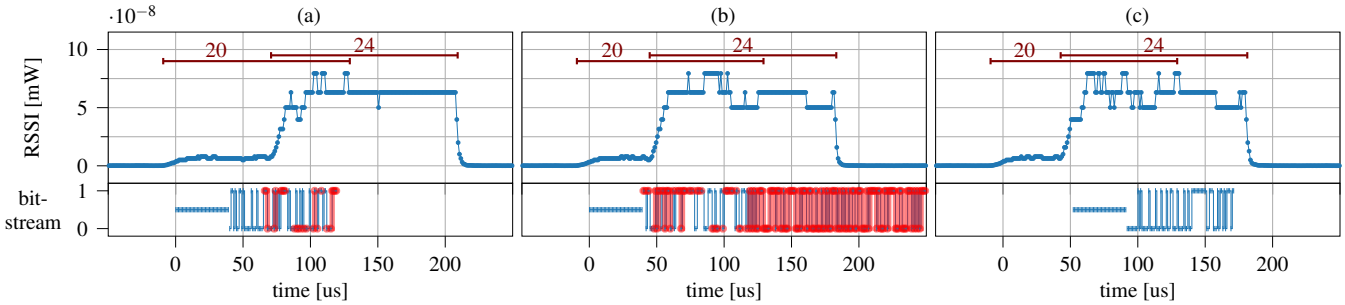

Figure 6: Capture effect. *Transmissions of nodes 20 and 24 arriving at node 3 ($f_c = 2483\,MHz$). Receive power of node 24 is about 10x (10 dB) higher compared to node 20. The transmit activity markers are aligned with the bitstream and extended by 10 µs at each side to include radio ramp-up and ramp-down times. (a) Strong packet starts 80 µs after weak packet and corrupts received bits. (b) Strong packet starts 54 µs after weak packet. Amongst others it corrupts the length field, so the receiver tries to demodulate more bits than sent. (c) Strong packet starts 52 µs after weak packet. The strong signal corrupts the weak packet's access address and inhibits the detection of a valid synchronization header. In consequence, the receiver continues scanning and locks on the strong packet, which it receives successfully.*

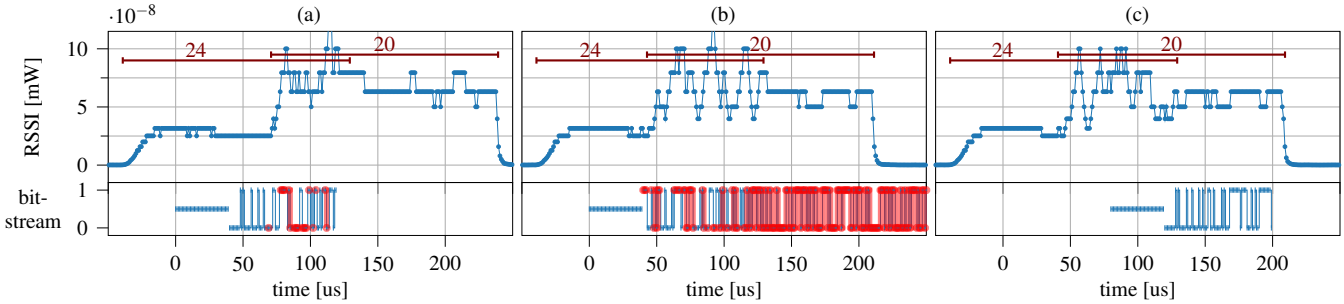

Figure 7: Capture effect with 30 µs pure carrier signal before the first bit. *Transmissions of nodes 20 and 24 arriving at node 3 ($f_c = 2401\,MHz$). Receive power of node 20 is about 2.5x (4 dB) higher compared to node 24. (a) Strong packet starts 110 µs after weak packet. (b) Strong packet starts 82 µs after weak packet. (c) Strong packet starts 80 µs after weak packet. The receiver captures the strong packet with 80 µs delay, i.e., the inserted pure carrier period effectively extends the capture window.*

interfering signal from node 7 does not break a running reception from node 24, but the other way around node 7 is not received successfully when disturbed by node 24. Nodes 2 and 8 are received at node 1 with a relatively small power difference. Here, interference is destructive in both cases, i.e., even if the interfering signal is weaker, its disturbing impact is strong enough to hinder successful reception. Thus, the experimental results confirm the analytical expectations.

4.4 Capture Effect

In the following experiment we investigate the well-known *capture effect* [23]. In essence, it reflects the following question: In view of Sec. 4.3, is it possible that the receiver turns over to a strong packet if it starts later than a weak one (instead of continuing a foredoomed reception)? The known answer is yes, but only if the strong transmission starts within a specific time frame T_C, i.e., with a bounded offset relative to the weak packet. This time frame, termed *capture window*, represents a synchronization interval that is needed by the receiver to detect a packet header. Once a valid header has been detected, the receiver stops scanning for other packets and goes with the detected transmission. As we shall see, exploring the capture effect with RssiSpy reveals some very interesting results.

Scenario. Node A starts a transmission. A listening node observes the transmitted signal and tries to synchronize. A stronger node B starts another transmission with time offset t_C relative to node A.

Expectation. Provided that the power difference between nodes A and B is high enough (compare Sec. 4.3), there are two cases. If $t_C > T_C$, then reception should fail. If $t_C < T_C$, then the receiver should successfully receive the packet from node B. The BLE 1M link layer packet starts with a constant 8 bit preamble followed by a 32 bit address field (Fig. 4a). So far it is unclear if the address field has influence on the capture window, so T_C is expected to equal 8 µs or 40 µs.

Experiment. We perform the experiment with nodes 20 and 24 transmitting to node 3 (Fig. 3). As in Sec. 4.3, nodes A and B first transmit separate packets that are used to estimate the receive power of both nodes. Afterwards, both nodes transmit overlapping packets with different payloads, where node 24 starts t_C after node 20. We test different values of t_C.

Results. The results are shown in Fig. 6. With $t_C = 80\,\mu s$ the receiver locks onto the weak signal and starts reception. From the point of its appearance, the strong signal causes bit errors and corrupts the received packet. With $t_C = 54\,\mu s$ the strong signal appears earlier and damages also the packet's length field, which makes the receiver trying to demodulate more bits than sent. In contrast, with $t_C = 52\,\mu s$ the receiver does not lock onto the weak signal and instead receives the strong

packet. Apparently, the strong signal starts early enough to prevent the detection of a valid packet header in the weak signal. For the specific test run we can conclude that $52\,\mu s \leq T_C \leq 54\,\mu s$. In our experiments the critical value of t_C varies slightly. Combining all our runs we observed $48\,\mu s < T_C < 55\,\mu s$ (without claiming any statistical reliability).

This result is interesting in two respects. First, it reveals that the capture window T_C includes the address field, at least on the nRF52840. Second, the capture window is still longer than the expected $40\,\mu s$. The reason for the latter becomes visible thanks to RSSISPY. The disruptive effect of the strong signal does not start with the packet's first bit, it starts immediately when signal power increases, i.e., during the strong transmitter's radio ramp-up phase. In Fig. 6c it is clearly visible how the power of the strong signal rises during the radio ramp-up and starts to dominate the receive signal about $10\,\mu s$ before the first bit.

This observation provokes an interesting question: Is it possible to increase the capture window by artificially extending the delay between enabling the radio's transmit path (power amplifier) and transmission of the first bit? To answer this question, we modify the experiment in that we start each transmission with a $30\,\mu s$ pure carrier signal before the first preamble bit. The results of this modified experiment are show in Fig. 7. It turns out that artificially delaying the packet header indeed increases the capture window by the same length, as the inserted signal segment acts like a header extension. In other words, using this trick it is possible to realize an arbitrarily large capture window and increased timing tolerance—for the price of longer packet air times, higher energy consumption, and wasted channel capacity. Note that the operational principle is not specific for BLE 1M or nRF52840, it can be implemented on every radio platform that allows to inject some kind of "garbage" before the packet.

4.5 Synchronous Transmissions

Next, we focus on the special case of *synchronous transmissions*. In this context, "synchronous" means that all transmitters *send identical data at (almost) the same time*, i.e., under idealized assumptions[7] we have $\tau_k = \tau$ and $u_k(t) = u(t)$ for all k. Note that there is no oscillator synchronization, so we still have to cope with CFO (Δf_c) and phase differences ($\Delta\phi_0$). The main motivation that pushed research on ST has been the believe that ST enable *constructive interference*. We use RSSISPY to have a very close look at this topic. Before we discuss our experiments, we derive the power of the receive signal analytically to inform our expectations.

4.5.1 Receive Signal Strength with ST

With $\tau_k = \tau$ and $u_k(t) = u(t)$, (8) can be rewritten as

$$v(t) = u(t-\tau) \underbrace{\sum_{k=1}^{N} h_k e^{\mathrm{j}2\pi\Delta f_{c,k} t}}_{c(t)} \tag{9}$$

i.e., the downconverted signal v equals the delayed baseband signal u modulated by the term $c(t)$. Although the CFOs $\Delta f_{c,k} = f_{c,k} - f_{c,R}$ depend on $f_{c,R}$, their values are different in general, and the receiver is unable to compensate for all of them with a single choice of $f_{c,R}$. Hence, $c(t)$ introduces a time-varying gain, which leads to a fluctuating receive signal strength. In simple cases this fluctuation has a characteristic sinusoidal pattern, so the phenomenon is known as the *beating effect* [16]. For the general case, the magnitude of $c(t)$ can be derived as follows (c^* denotes the complex conjugate of c).

$$\begin{aligned}
|c(t)|^2 = c(t)\,c^*(t) &= \Big(\sum_k h_k e^{\mathrm{j}2\pi\Delta f_{c,k} t}\Big)\Big(\sum_k h_k^* e^{-\mathrm{j}2\pi\Delta f_{c,k} t}\Big) \\
&= \sum_k \sum_l |h_k||h_l| e^{\mathrm{j}(2\pi\Delta f_{c,k} t + \Delta\phi_{0,k})} e^{-\mathrm{j}(2\pi\Delta f_{c,l} t + \Delta\phi_{0,l})} \\
&= \sum_k |h_k|^2 + \sum_k \sum_{l>k} |h_k||h_l| \big(\qquad (*) \\
&\qquad\qquad e^{+\mathrm{j}(2\pi(\Delta f_{c,k} - \Delta f_{c,l})t + \Delta\phi_{0,k} - \Delta\phi_{0,l})} + \\
&\qquad\qquad e^{-\mathrm{j}(2\pi(\Delta f_{c,k} - \Delta f_{c,l})t + \Delta\phi_{0,k} - \Delta\phi_{0,l})} \big) \\
&= \sum_{k=1}^{N} |h_k|^2 + 2 \sum_{k=1}^{N} \sum_{l=k+1}^{N} |h_k||h_l| \cdot \qquad (10) \\
&\qquad\qquad \cos\big(2\pi(\Delta f_{c,k} - \Delta f_{c,l})t + \Delta\phi_{0,k} - \Delta\phi_{0,l}\big)
\end{aligned}$$

Here, $(*)$ exploits the symmetry of the second line (it combines the addends (k,l) and (l,k)), while (10) follows from the rule $\cos\phi = (e^{\mathrm{j}\phi} + e^{-\mathrm{j}\phi})/2$. In essence, (10) shows that the *mean* receive signal strength is the sum of the individually attenuated transmit powers, while the *instantaneous* receive signal strength is a mixture of sinusoidal waves that fluctuate around the mean. Overall, it holds $0 \leq |c(t)|^2 \leq (\sum_k |h_k|)^2$.

4.5.2 ST with Two Transmitters

We begin our ST experiments with $N = 2$ transmitters and use RSSISPY to examine the resulting beating patterns (i.e., the shape of $|c(t)|^2$) as well as the position of bit errors. We expose the correlation between the two and determine intervals with constructive ($|c(t)|^2 > |h_k|^2$) and destructive ($|c(t)|^2 < |h_k|^2$) interference.

Scenario. Two nodes A and B transmit a packet synchronously (i.e., identical packet at exactly the same time). A listening node tries to receive the overlayed packet. We focus on constellations where A and B alone are received with similar power, i.e., $|h_A| \approx |h_B|$.

Expectation. With $N = 2$, (10) reads as

$$\begin{aligned}
|c(t)|^2 = |h_A|^2 + |h_B|^2 + 2|h_A||h_B| \cdot \\
\cos(2\pi(\Delta f_{c,A} - \Delta f_{c,B})t + \Delta\phi_{0,A} - \Delta\phi_{0,B})
\end{aligned}$$

Given $|h_A| \approx |h_B|$ this simplifies to $|c(t)|^2 \approx 2|h_A|^2\big(1 + \cos(\ldots)\big)$, i.e., the receive power should have a sinusoidal shape and swing between 0 and $4|h_A|^2$.

Experiment. We perform the experiment with nodes 9 and 24 transmitting to node 3 and with nodes 2 and 8 transmitting to node 1 (Fig. 3). As in Sec. 4.3, nodes A and B first transmit separate packets that are used to estimate the receive power of both nodes. In addition, in the beginning of each experiment we use timestamps of such packets to determine the oscillator

[7] The scenario is most interesting if multiple transmitters have similar receive power, as otherwise the dominant transmitter prevails as in Sec. 4.3. For equally strong transmitters it is assumed that the differences in time-of-flight are negligible compared to the symbol duration. Further, modulator imperfections other than CFO and phase drift are neglected.

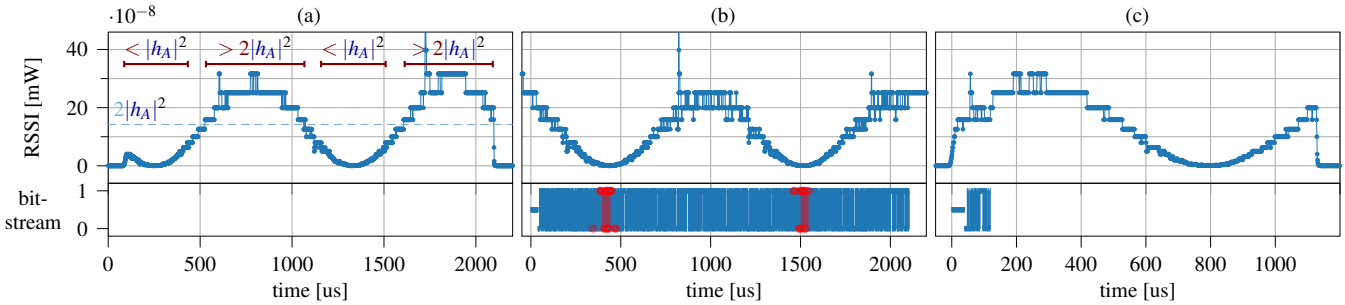

Figure 8: Synchronous transmissions with slow beating. *Transmissions of nodes 9 and 24 arriving at node 3 ($f_c = 2483$ MHz). Receive power of both nodes ($|h_A|^2$) is about $7 \cdot 10^{-8}$ mW (-71.5 dBm). The relative oscillator drifts have been measured as -5.0 ppm and -5.4 ppm, respectively. (a) Simultaneous transmission of an unmodulated carrier signal. The CFOs of the transmitting nodes cause a sinusoidal receive signal strength with phases of constructive and destructive interference. The beat frequency is close to 1 kHz (one period per 1000 µs). (b) Simultaneous transmission of two long identical packets. Destructive interference causes bit errors whenever the receive signal strength falls below a critical level. (c) Simultaneous transmission of two short identical packets. The packet is received without errors, as it lies in a phase with constructive interference.*

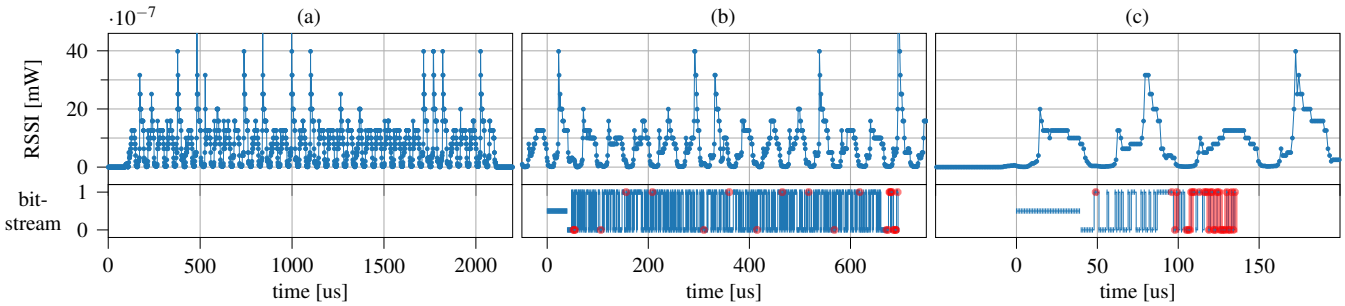

Figure 9: Synchronous transmissions with fast beating. *Transmissions of nodes 2 and 8 arriving at node 1 ($f_c = 2401$ MHz). Receive power of both nodes is $4 \cdot 10^{-7}$ mW (-64 dBm). The relative oscillator drifts have been measured as +7.7 ppm and -0.4 ppm, respectively. (a) Simultaneous transmission of an unmodulated carrier signal. The CFOs cause a sinusoidal receive signal strength. The beat frequency is close to 19 kHz (one period per 53 µs). (b), (c) Simultaneous transmission of long (b) and short (c) identical packets. Destructive interference causes bit errors whenever the receive signal strength falls below a critical level. The receiver tries to demodulate a wrong number of bits because the length fields are corrupted.*

drift between each transmitter and the receiver and, based on that, to estimate the CFOs $\Delta f_{c,A}$ and $\Delta f_{c,B}$. Eventually, nodes A and B synchronously transmit (i) a pure carrier signal, (ii) a short packet followed by a long pure carrier period, and (iii) a long packet surrounded by pure carrier periods. The carrier periods are used to reveal the beating patterns more clearly, which is otherwise impossible with short packet air times.

Results. The results of the first experiment are shown in Fig. 8. The receive signal strength has a sinusoidal shape and swings between 0 and roughly $25 \cdot 10^{-8}$ mW $\approx 4|h_A|^2$ (remember the nonlinear granularity of the samples). Based on the measured oscillator drifts, the expected beat frequency is $\Delta f_{c,A} - \Delta f_{c,B} = (\text{-}5.0\,\text{ppm} + 5.4\,\text{ppm}) \cdot 2483\,\text{MHz} = 993\,\text{Hz}$, which is surprisingly close to the observed 1 kHz (one beat per 1000 µs). Bit errors arise whenever the receive signal strength falls below a critical threshold. The beat period is long enough to enable a successful transmission of some consecutive (short) packets with a certain probability. This case is termed *slow beating* (aka *wide* or *long* beating) [16, 8, 7].

In contrast, the results of the second experiment shown in Fig. 9 illustrate *fast* (or *narrow*) *beating*. Here, the expected beat frequency is $\Delta f_{c,A} - \Delta f_{c,B} = (7.7\,\text{ppm} + 0.4\,\text{ppm}) \cdot 2401\,\text{MHz} = 19.4\,\text{kHz}$, which is again close to the observed value of 19 kHz (one beat per 53 µs). The power drops occur more frequently (with shorter duration), making it impossible to successfully transmit even short packets.

4.5.3 ST with Three Transmitters

In this experiment we increase the number of transmitters to $N = 3$ and again examine the beating patterns with the help of RssiSpy. The example illustrates the intricate beating shapes resulting from a larger number of transmitters.

Scenario. We straightforwardly extend the setting of Sec. 4.5.2 to three transmitters A, B, and C, which send an identical packet at the same time. Again, a listening node tries to receive the overlayed packet.

Expectation. With $N = 3$ we can rewrite (10) as

$$\begin{aligned} |c(t)|^2 = {} & |h_A|^2 + |h_B|^2 + |h_C|^2 \\ & + |h_A||h_B|\cos(2\pi(\Delta f_{c,A} - \Delta f_{c,B})t + \Delta\phi_{0,A} - \Delta\phi_{0,B}) \\ & + |h_A||h_C|\cos(2\pi(\Delta f_{c,A} - \Delta f_{c,C})t + \Delta\phi_{0,A} - \Delta\phi_{0,C}) \\ & + |h_B||h_C|\cos(2\pi(\Delta f_{c,B} - \Delta f_{c,C})t + \Delta\phi_{0,B} - \Delta\phi_{0,C}) \end{aligned}$$

Hence, if all pairwise CFO differences are unequal, then the receive power's shape should appear as a mixture of three sinusoidal signals. If any differences are equal, then the mixture should seem to contain accordingly less components.

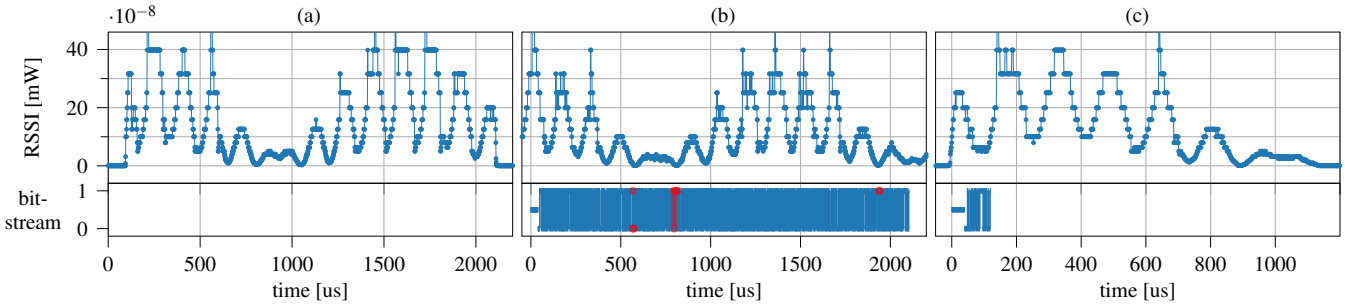

Figure 10: Synchronous transmissions with three transmitters. *Transmissions of nodes 9, 20, 24 arriving at node 3 (f_c = 2483 MHz). Receive power of nodes 9 and 24 is about $6.3 \cdot 10^{-8}$ mW (-72 dBm), node 20 is received with $2.5 \cdot 10^{-8}$ mW (-76 dBm). The relative oscillator drifts have been measured as -5.0 ppm, -2.6 ppm, and -5.3 ppm, respectively. (a) Simultaneous transmission of an unmodulated carrier signal. The receive signal strength caused by the transmitter's CFOs is a superposition of multiple sine waves. Two beat frequencies can be identified: about 700 Hz (one period per 1400 µs), and 6.7 kHz (one period per 150 µs). (b) Simultaneous transmission of three long identical packets. Destructive interference causes bit errors whenever the receive signal strength falls below a critical level. (c) Simultaneous transmission of three short identical packets. The packet is received without errors, the receive power is high enough for successful reception.*

Experiment. We perform the experiment with nodes 9, 20, and 24 transmitting to node 3 (Fig. 3). As in Sec. 4.5.2, we use separate packets to estimate the receive power and oscillator drift of each node. Then, all three nodes synchronously transmit the same test signals as in Sec. 4.5.2.

Results. The results are shown in Fig. 10. The receive signal strength is a mixture of sine waves, which has a more complex shape compared to the two-transmitter case (Sec. 4.5.2). Based on the measured oscillator drifts, there should be three components with frequencies 745 Hz, 6.0 kHz, and 6.7 kHz. Two of them are clearly visible, but the 6.0 kHz component cannot be identified. To clarify what is happening here, we conducted another experiment where the same nodes transmit a long carrier signal (25 ms). We analyzed the recorded results via Fourier transform. The frequency spectrum indeed showed three significant components at 710 Hz, 5.95 kHz, and 6.66 kHz. It turns out that, except for the trivial cases, manually interpreting beating patterns is an error-prone task.

Besides that, bit errors again arise when the receive signal strength falls below a critical threshold. It is interesting that only the deepest power drops cause bit errors in the concrete experiment. This cannot be generalized. It underlines the diversity of beating patterns and their consequences, and the difficulty in predicting individual effects in larger networks.

4.6 Other Insights

Before we conclude our experiments, we want to mention some observations that are of general interest and independent of specific concurrent transmission scenarios.

Overshoot and AGC influence. In some of our measurements we find overshoot effects in form of singular spikes (e.g., in Fig. 8b) or systematic patterns (more or less, e.g., in Fig. 9). We suppose that this is related to the AGC as follows. Let P_a denote the power of the receive signal v after amplification, i.e., $P_a(t) = G \cdot |v(t)|^2$. The task of a typical AGC is to steer the gain G such that P_a reaches some fixed constant value, i.e., $P_a(t) = G(t) \cdot |v(t)|^2 = const$. Hence, $G(t)$ can be used as an RSSI proxy because under ideal conditions it is inversely proportional to $|v(t)|^2$. However, the system dynamics of a real AGC control loop are not ideal, and utilizing $G(t)$ as RSSI causes artifacts. We believe that this is the reason for the following observations found in our experiments, all with multiple transmitters active: (i) significant overshoot in Figures 5 and 7, (ii) single outliers in Fig. 8, (iii) overshoot beyond $4|h_A|^2 = 16 \cdot 10^{-7}$ mW in Fig. 9 and similar overshoot in Fig. 10. Further, Fig. 8c shows an interesting effect that substantiates our suspicion that the nRF52840's RSSI output is influenced by the AGC: While the RSSI signal has the expected sinusoidal shape outside of the packet, it exhibits a remarkably stable level during the packet transfer. All these effects emphasize the importance of considering the AGC. Nonetheless, we are confident that the resulting artifacts do not diminish the utility of RSSISPY.

Radio event timing. Let T_S denote the delay between starting a packet transmission (after radio ramp-up) and the first preamble bit on air. Further, let T_R be the delay between the first preamble bit on air and the generation of a corresponding radio event at the receiver. Ideally, one would expect $T_S = \varepsilon_S$ and $T_R = \tau + \varepsilon_R$, where $\varepsilon_x \ll 1\,\mu$s represents some small internal processing delay. In practice, however, we find that $T_S + T_R \approx 10\,\mu$s in BLE 1M mode, i.e., the nRF52840's radio induces delays on the order of multiple bit times. This leads to a tricky problem because without additional measuring equipment it is impossible to split the sum value into T_S and T_R (any measurement from start to event includes both, T_S and T_R). Hence, there is an uncertainty of several microseconds in the temporal alignment of the bitstreams.[8] We decided to pretend $T_S = 0$, i.e., our plots show each bitstream at the earliest possible position. Comparing the location of bit errors with the RSSI signals (e.g., in Fig. 6) suggests that the actual value of T_S is greater, i.e., the true position of each bitstream is a few microseconds to the right. However, determining the precise values of T_S and T_R is beyond the scope of this paper, and in favor of clarity we avoided the usage of vague estimations.

5 Discussion

Other radio standards. In this paper we focus on the BLE 1M mode. Using RSSISPY with other BLE modes is straight-

[8] This issue does not impair clock synchronization because the latter relies mostly on the sum $T_S + T_R$ and does not require to split the terms apart.

forward, essentially it only requires the adaptation of some configuration registers and constants. It is an interesting question if RSSISPY can also be used with other narrowband radio standards, e.g., IEEE 802.15.4. The used RSSI sampling functionality of the nRF52840 is assigned to BLE [6], but maybe it also works in 15.4 mode. Besides that, even in BLE mode RSSISPY detects any energy in the selected channel (BLE signals, interference, noise), i.e., it can be used as a "power meter" for all kinds of radio signals in the selected frequency band. Indeed, we demonstrate this capability with pure carrier signals, which are examples for non-BLE signals.

Other applications of RSSISPY. In the shown experiments we process the RSSI samples *offline* and use them to manually inspect and analyze specific interference scenarios. The continuous sample streams also facilitate a more reliable estimation of important parameters like link power and SNR. That said, we believe that the possibility to use RSSISPY "in the wild", i.e., in the same nodes and running in parallel to the application of interest, enables a new class of protocols that exploit RSSI streams *online*. For example, the characteristics of beating patterns presented in Sec. 4.5 could render it possible to estimate the number of transmitters and their parameters (e.g., $|h_k|$, $\Delta f_{c,k}$, $\Delta\phi_{0,k}$) from a single transmission. This side information could be used in higher protocol layers to adapt traffic patterns and make communication more efficient. It may also allow to decode information from erroneous packets that are otherwise useless. Further, it is possible to encode information in the RSSI signal itself, which is a common method to enable CTC [13]. The high sample rate provided by RSSISPY could raise CTC capabilities to a new level. It may also be beneficial for localization and ranging tasks, e.g., to cope with very fast moving nodes.

Sample rate. The ideal sample rate is determined by the highest signal frequency or the anti-aliasing filter. Unfortunately, factors like external interference can induce high frequency components, and little is known about nRF52840's filter. So, RSSISPY provides a very high sample rate to overcome potential issues. In the rare case that the highest possible frequency is known in advance, the sample rate may be reduced, e.g., to enable the implementation of an extended trigger logic.

6 Conclusions

We have presented RSSISPY, a software tool for continuous RSSI sampling on COTS devices. Compared to prior solutions, RSSISPY provides more than 20× higher time resolution and enables bit-level RSSI inspection in the target environment, which empowers research and inspires new applications. We used RSSISPY to investigate concurrent transmissions in low-power wireless networks and uncovered important details of physical layer effects that have been unknown so far. Beyond that, we have generalized the analytical description of beating patterns in ST for an arbitrary number of transmitters and validated this result in experiments.

7 Acknowledgments

This work was supported by the German Research Foundation (DFG) within the Emmy Noether project NextIoT (grant ZI 1635/2-1).

8 References

[1] Describe late-arriving interrupt behaviour. Knowledge Base Article KA001190, Arm Limited.

[2] Cortex-M4 Devices. Generic User Guide ARM DUI 0553B (ID012616), ARM, Aug. 2011.

[3] Cortex-M4(F) Lazy Stacking and Context Switching. Application Note 298, ARM DAI0298A (ID032612), ARM Limited, Mar. 2012.

[4] ARM Cortex-M4 Processor. Technical Reference Manual Revision r0p1, 100166_0001_04_en, Arm Limited, May 2020.

[5] ARM v7-M. Architecture Reference Manual ARM DDI 0403E.e (ID021621), Arm Limited, Feb. 2021.

[6] nRF52840. Product Spec. 4413_417 v1.7, Nordic Sem., Nov. 2021.

[7] B. Al Nahas, A. Escobar-Molero, J. Klaue, S. Duquennoy, and O. Landsiedel. BlueFlood: Concurrent Transmissions for Multi-Hop Bluetooth 5 – Modeling and Evaluation. *arXiv:2002.12906*, Apr. 2021.

[8] M. Baddeley, C. A. Boano, A. Escobar-Molero, Y. Liu, X. Ma, U. Raza, K. Römer, M. Schuß, and A. Stanoev. The Impact of the Physical Layer on the Performance of Concurrent Transmissions. In *IEEE Int. Conf. Network Protocols*, ICNP '20, pages 1–12, Madrid, Spain, Oct. 2020.

[9] C. A. Boano, M. Schuß, and K. U. Römer. EWSN Dependability Competition: Experiences and Lessons Learned. *IEEE Internet of Things Newsletter*, Mar. 2017.

[10] K. Chebrolu and A. Dhekne. Esense: communication through energy sensing. In *MobiCom '09*, pages 85–96, New York, USA, Sept. 2009.

[11] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh. Efficient network flooding and time synchronization with Glossy. In *IPSN '11*, pages 73–84, Chicago, IL, USA, Apr. 2011.

[12] X. Guo, Y. He, and X. Zheng. WiZig: Cross-Technology Energy Communication Over a Noisy Channel. *IEEE/ACM Transactions on Networking*, 28(6):2449–2460, Dec. 2020.

[13] Y. He, X. Guo, X. Zheng, Z. Yu, J. Zhang, H. Jiang, X. Na, and J. Zhang. Cross-Technology Communication for the Internet of Things: A Survey. *ACM Computing Surveys*, Mar. 2022. Just Accepted.

[14] R. Hofmann, C. A. Boano, and K. Römer. X-Burst: Enabling Multi-Platform Cross-Technology Communication between Constrained IoT Devices. In *SECON '19*, pages 1–9, Boston, MA, USA, June 2019.

[15] S. M. Kim and T. He. FreeBee: Cross-technology Communication via Free Side-channel. In *MobiCom '15*, pages 317–330, New York, USA, Sept. 2015.

[16] C.-H. Liao, Y. Katsumata, M. Suzuki, and H. Morikawa. Revisiting the So-Called Constructive Interference in Concurrent Transmission. In *LCN '16*, pages 280–288, Dubai, United Arab Emirates, Nov. 2016.

[17] C. Noda, C. M. Prez-Penichet, B. Seeber, M. Zennaro, M. Alves, and A. Moreira. On the Scalability of Constructive Interference in Low-Power Wireless Networks. In T. Abdelzaher, N. Pereira, and E. Tovar, editors, *Wireless Sensor Networks*, pages 250–257, Cham, 2015. Springer International Publishing.

[18] V. S. Rao, M. Koppal, R. V. Prasad, T. V. Prabhakar, C. Sarkar, and I. Niemegeers. Murphy loves CI: Unfolding and improving constructive interference in WSNs. In *INFOCOM '16*, pages 1–9, San Francisco, CA, USA, Apr. 2016.

[19] B. Sklar and F. J. Harris. *Digital Communications: Fundamentals and Applications, 3rd Edition*. Prentice Hall, Harlow, Dec. 2020.

[20] R. Trüb, R. Da Forno, L. Sigrist, L. Mühlebach, A. Biri, J. Beutel, and L. Thiele. FlockLab 2: Multi-Modal Testing and Validation for Wireless IoT. In *CPS-IoTBench '20*, pages 1–7, Sept. 2020.

[21] Y. Wang, Y. He, X. Mao, Y. Liu, and X.-y. Li. Exploiting Constructive Interference for Scalable Flooding in Wireless Networks. *IEEE/ACM Transactions on Networking*, 21(6):1880–1889, Dec. 2013.

[22] M. Wilhelm, V. Lenders, and J. B. Schmitt. On the Reception of Concurrent Transmissions in Wireless Sensor Networks. *IEEE Transactions on Wireless Communications*, 13(12):6756–6767, Dec. 2014.

[23] D. Yuan and M. Hollick. Let's talk together: Understanding concurrent transmission in wireless sensor networks. In *LCN '13*, pages 219–227, Sydney, Australia, Oct. 2013.

[24] M. Zimmerling, L. Mottola, and S. Santini. Synchronous Transmissions in Low-Power Wireless: A Survey of Communication Protocols and Network Services. *ACM Computing Surveys*, 53(6):1–39, Nov. 2021.