A Flexible Framework for Debugging IoT Wireless Applications

Citation for published version:
Gringoli, F, Ali, N, Guerrini, F & Patras, P 2018, A Flexible Framework for Debugging IoT Wireless Applications. in Proceedings for the IEEE International Workshop on Metrology for Industry 4.0 & IoT. IEEE, Brescia, Italy, pp. 230-235, 2018 IEEE international workshop on Metrology for Industry 4.0 & IoT, Brescia, Italy, 16/04/18. DOI: 10.1109/METROI4.2018.8428337

Digital Object Identifier (DOI):
10.1109/METROI4.2018.8428337

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings for the IEEE International Workshop on Metrology for Industry 4.0 &amp; IoT

A Flexible Framework for Debugging IoT Wireless Applications

Francesco Gringoli, Nahla Ali, Fabrizio Guerrini
Dept. of Information Engineering
University of Brescia Brescia, Italy
<*name.surname*>@unibs.it

Paul Patras
School of Informatics
University of Edinburgh Edinburgh, Scotland
ppatras@inf.ed.ac.uk

***Abstract*—Debugging IoT wireless applications can be a tough task. Different communication protocols may simultaneously operate in the same RF band, giving rise to ambiguities when trying to understand in which order frames are transmitted by the same application or when frames are affected by errors. In this paper we present a flexible framework for capturing all types of communication taking place in the 2.4GHz band, irrespective of the governing standard. As an example, we demonstrate accurate sniffing and time-stamping of Wi-Fi and Bluetooth Low Energy frames.**

***Keywords*-IoT debugging, protocol sniffing.**

I. Introduction

Debugging IoT wireless applications can be very difficult, as multiple communication sessions following different standards may be employed at the same time in the same frequency band. Several frameworks for capturing network traffic that complies with specific standards already exist. Combining data from different sniffers is however challenging, especially when timing is crucial, as for instance when synchronising traces is only possible with the support of an accurate external time reference. The lacking of such precision leads to ambiguities when aiming to explain a sequence of actions or more easily which frames are colliding. Professional equipment that solve this issue have recently emerged [1], yet they are expensive and not upgradeable with software written by users, i.e. only proprietary software can be used to update their decoding capabilities. Open-source alternatives are largely tied to non-flexible hardware designed for a specific standard [3] and work with closed-source firmwares in the lower layers [2]–[4]. Furthermore, these can only capture traffic on a single channel at a time and, in the case of Bluetooth, they are unable to follow a conversation, unless an initial connection request was intercepted. Similarly, Software-Defined Radio (SDR) solutions have been proposed to decode one particular standard over a fixed channel [8]. With respect to Bluetooth Low Energy (BLE) many papers tried to deal with the security of its communications, i.e. try to intercept an ongoing communication and decrypt it without any knowledge of the initial connection establishment [5]–[7], while we are instead interested in debugging communication sessions.

To overcome these limitations, in this paper we propose a flexible framework for i) capturing the entire activity in the 2.4GHz band, across a total spectral width of 80MHz, using multiple SDRs with limited receiving bandwidth and ii) extracting frames originating from transmitters that follow different standards and time-stamping these using a common time horizon. To demonstrate the capabilities of the proposed framework, we focus on a scenario where 802.11g and BLE frames are simultaneously present on a channel. We construct specific use cases and present preliminary results confirming that our solution provides very accurate time-stamps; these can be used to perfectly understand frame transmission order, which can drastically simplify wireless protocol debugging in crowded IoT environments.

The rest of the paper is organised as follows. We introduce our capturing architecture in Section II, the decoding engine in Section III, we report preliminary results on capturing BLE and Wi-Fi frames in Section IV, and we conclude the paper in Section V.

II. Debugging Framework

The hardware setup underpinning the proposed wireless debugging infrastructure is illustrated in Fig. 1. We use two Ettus B200 SDR boards [9] for signal capture, which we connect to a mid-end workstation powered by an Intel Core-i7 CPU with four cores clocked at 4.2GHz, and equipped with 16GB of DDR4-3000 RAM. The system runs Ubuntu 16.04 LTS. The two SDR devices support full-duplex operation with up to 56 MHz of real-time bandwidth. To avoid USB level bottlenecks, we connect these front ends to the host through two separate USB-3.0 controllers plugged into the PCI-E slots of an Asus Z270-A motherboard. The controlling hosts employs a Samsung 850 EVO SATA III m.2 solid state drives (SSD) for storing the captured traces.

A. Capturing Architecture

We are interested in capturing network traffic on all BLE and 802.11g channels, as defined by the respective standards [10], [11]. The carrier frequencies of the corresponding channels can be expressed as follows

$$f_k^{\text{BLE}} = 2,402\text{MHz} + k \cdot 2\text{MHz}, k \in [0, 39], \quad (1)$$
$$f_n^{\text{Wi-Fi}} = 2,407\text{MHz} + n \cdot 5\text{MHz}, n \in [1, 13]. \quad (2)$$

Since BLE channels are 2MHz-wide and respectively span 20/22MHz in the case of Wi-Fi (for both 802.11g and 802.11b encodings), we need to capture the frequency spectrum ranging

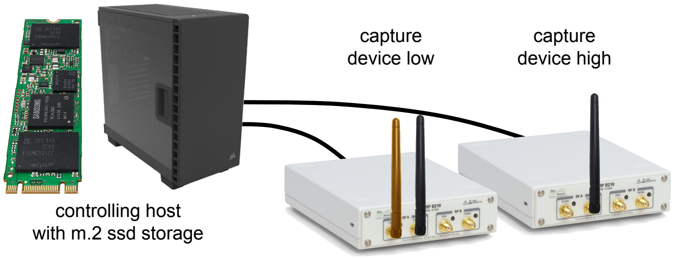

Fig. 1. Capturing platform built with a pair of Ettus B200 SDRs connected to the same controlling host. The lightly coloured antenna of the device on the left is used to transmit a periodic synchronisation signal.

from 2,401MHz to 2,483MHz. We note that the leftmost margin is shared by BLE channel 37[1] and 802.11g channel 1. The rightmost margin covers 802.11g channel 13, whilst the rightmost BLE channel is located 3MHz below.

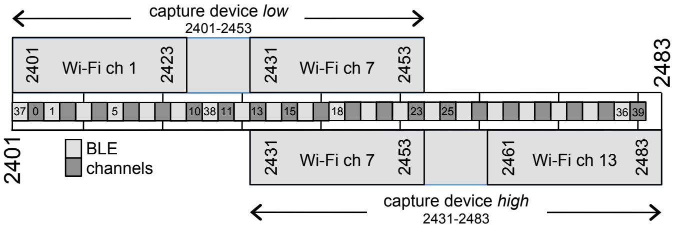

Fig. 2. ISM band covered by the designed framework. Wi-Fi channels 1–13 and all BLE channels are covered by the two capture devices.

To cover the whole spectrum, we configure the two capture chains as shown in Fig. 2. We set the signal sampling frequency to 52MS/sec and tune the carrier of the *capture device low* to 2,427MHz and that of *capture device high* to 2,457MHz. Even though it may not seem strictly necessary, the large 22MHz overlap between the two devices brings two main benefits: i) it allows to work with the same sampling rate, so that processing samples from the two captured traces can be synchronous, and ii) it enables in principle to share an entire Wi-Fi channel and use this to synchronise the two traces during post-processing. Besides, only 5MHz of spectrum could be saved from either the low or the high capture chain. That is we could only move to the right by 5MHz the left margin of the right capture chain, as we need to capture Wi-Fi channel 8. In addition, this configuration is amenable to also capturing 40MHz-wide channels in any configuration. It should be hence straightforward to extend our framework to enable 802.11n decoding, which we leave for future work.

Due to their spectral overlap, we do not even need the two capture chains to be coherent, i.e. they need not be fed with an external clock reference. This makes the testbed setup considerably simpler. Instead, we synchronise during post-processing the traces acquired, when extracting samples from both. We use frames that we capture at both chains to synchronise the captures. In particular, we could opportunistically use Wi-Fi frames captured on channel 7 or BLE frames overheard on channels from 13 to 23. The main drawback of such an approach is the inability to record Wi-Fi frames potentially transmitted on channel 7. Likewise, we might not be able to decode BLE frames belonging to data sessions for which a connection request was not captured previously, as we discuss in detail in the next section. A more effective solution is to periodically transmit using capture device low a short BLE frame with known preamble and with an embedded counter to avoid synchronisation ambiguities. We perform this operation by default once per second on BLE channel 18.

The reasons for using two SDR devices with bandwidth constraints instead of a single device SDR solution capable of capturing at higher sampling rate are as follows. Alternatives such as Ettus X300, which supports an operation bandwidth up to 160MHz, cost as much as four B200 devices and while providing only the main logic board with ADC/DAC circuitry. Additional radio daughterboards are required, which further increase the platform cost. Such a solution further requires an external power supply and either a PCI-e connection or a 10Gb/s Ethernet interface to be able to stream samples to the host, which introduces portability limitations.

B. Storage Architecture

We store on SDD all the data produced by the capturing architecture, to be used for later processing and decoding. The total sustained throughput θ_{stream} of I/Q samples is:

$$\theta_{\text{stream}} = 2 \cdot 2 \cdot 52\text{MS/sec} \cdot 1\text{B/S} = 208\text{MB/sec},$$

where we considered the two capture devices, the two (I and Q) samples, the capturing sample rate and a single byte to represent a real sample.

According to the manufacturer's technical data sheet, the SSD supports a sustained writing throughput of 520MB/sec, which matches the required performance. Nevertheless, we characterised the maximum sustained write speed obtainable under several configurations, to find the one that enables collecting very long traces without the risk of temporary loss, which would make the two captured traces desynchronise. To find the optimal setting we tested the following configurations:

- data stored on the same partition as the location of the Operating System (OS) – this is the worst case, as the OS is accessing the same partition for operations including logging, and periodic activity such as starting processes, etc.;
- data stored on a dedicated partition on the same drive as the OS – in this case the capturing process is the only one accessing the file system, but the device is still shared with the OS. We formatted the partition using either the `ext4` or `btrfs` file system; the latter should be faster, as it does not perform any journaling;
- data stored on a dedicated partition on a dedicated device – in this case the capturing process is the only one accessing the entire device. We formatted the partition using the best of previous configurations considered;

[1] Official channel numbers should not be confused with indexes k in Eq. 2

- raw access on a dedicated device – in this case the capturing process has exclusive access to the device and writes directly to the raw device as if it were a file.

To test access when writing to a partition we used `bonniee++`, a free file system benchmarking tool for Unix-like operating systems [12]. For the last option we developed a small piece of C code that writes 2GB of data divided into blocks of 20KB, the size of the captured blocks, and measures the time taken until the data is physically flushed to the drive. We report the results of our performance analysis in Table I.

Mode	Writing Speed (MB/sec)
Data to OS partition	340MB/sec
Data to dedicated `ext4` partition on same drive as OS	342MB/sec
Data to dedicated `btrfs` partition on same drive as OS	341MB/sec
Data to dedicated `ext4` partition on dedicated drive	342MB/sec
Raw access to dedicated drive	480MB/sec

TABLE I
Different storage options and achievable sustained write throughputs.

All storage options considered largely exceeded the required 208MB/s rate, therefore we decided to use for storage an `ext4` partition on the same drive as the OS, as this was easier to deploy than raw access. Interestingly, we note that using the `btrfs` file system led to similar performance to that of `ext4`, while using a dedicated drive did not bring any improvements. A notable boost appeared when we completely bypassed the file system, i.e. raw writing to the dedicated device. Specifically, this led to to a 1.4x speed improvement. We plan to use this option in a future release of the framework, for situations where we would need to capture even larger spectra.

Finally, we developed simple software to interface the two capture devices with the storage. The software starts two threads, one that is responsible with fetching the I/Q samples through the UHD library and storing them into RAM, and another that periodically dumps to storage the data collected. The software starts one additional thread that transmits every second a short BLE frame with known payload, which is later used to synchronise the captured traces.

III. Decoding engine

To process the traces saved by the two capturing devices we implemented the multi-protocol decoding tool whose operation we sketch in Fig. 3. This tool consists of a main loop that processes the data saved on the storage drive and feeds N individual decoders, which we create at start up. During an initial registration phase, each decoder requests a specific portion of spectrum by setting the central frequency, the bandwidth, and the expected sampling rate expressed as a rational number, $R_{\text{up}}/R_{\text{down}}$: here R_{up} is the up-sampling and R_{down} the down-sampling factors that will be used to reach any target rate that does not divide the original 52MS/sec. This creates a channeliser that later, during the decoding phase, shifts the spectrum of the sequence of samples to the configured central frequency, brings the sampling rate to the right value and applies a low-pass FIR filter for reducing the aliasing effect due to the up- and down-sampling operations. The framework automatically computes the number and values of the FIR tap coefficients. We illustrate this approach in Fig. 4. Which trace to use as input for each decoding chain is chosen during the registration phase, depending on the spectral characteristics requested by each decoder.

Signal processing is performed on a per-block basis. Each chain is fed with a block of N_{block} samples at a time. Because of the re-sampling, each corresponding decoder is fed with blocks of approximately $N_{\text{block}}R_{\text{up}}/R_{\text{down}}$ samples at a time and has to keep an internal state, since it is very unlikely that a frame will be completely within a single block of data. When a decoder successfully detects and decodes a packet, it can pass a string of data through a call-back mechanism. In this way it reports relevant information about the captured data to the trace collector, which saves this on storage. We plan to extend this functionality to add support for saving such captures as `pcap` traces.

A. Wi-Fi decoder

For the implementation of the 802.11g decoder we draw inspiration from the `gr-ieee802-11` framework [8], which was originally developed for GNURadio [14]. We re-implemented this decoding chain in our toolset, our prototype comprising blocks that process the incoming symbols and exploit their structure for i) *detecting* frames and ii) *receiving the data* they contain. Within communications following the 802.11g specification, frames are modulated using Orthogonal Frequency-Division Multiplexing (OFDM) where 20MHz of spectrum is divided into 64 carriers that encode symbols of given time duration. As we show in Fig. 5, all frames start with a Physical Layer Convergence Protocol (PLCP) header that is composed of a Short-Training-Sequence (STS), a Long-Training-Sequence (LTS) and a Signal symbol. All these fields encode data using a Binary-Phase-Shift-Keying

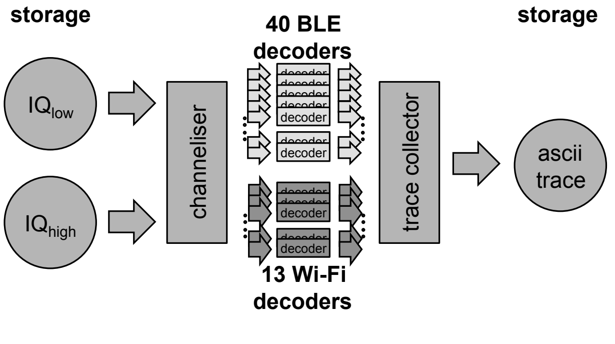

Fig. 3. Multi-protocol decoding software: a channeliser extracts several signal streams at the requested data-rate and centre frequency, then feeds each stream to separate serial decoders.

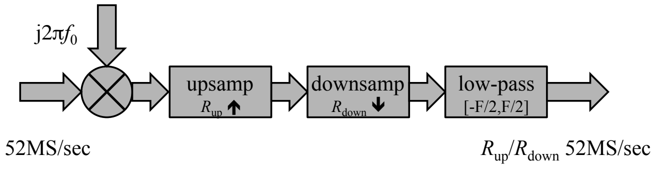

Fig. 4. Channeliser chain: each decoder receives the subset of samples corresponding to frequencies in range $[f_0 - F/2, f_0 + F/2]$ at rate $R_{\mathrm{up}}/R_{\mathrm{down}} \cdot 52\mathrm{MS/sec}$.

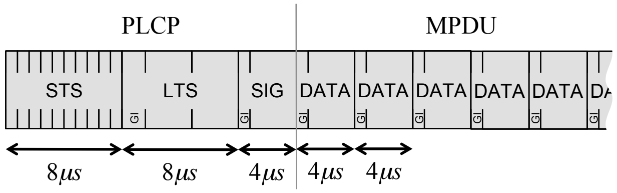

Fig. 5. The structure of an OFDM 802.11g frame.

constellation at each carrier. As such, we use these fields for frame detection, carrier frequency offset estimation, channel equalisation, and time synchronisation. The Signal symbol embeds information about the length of the following MAC Protocol Data Unit (MPDU) that carries user-data and its encoding, i.e. the mapping of bits to OFDM carriers. Thanks to this approach, frames having PLCP headers of the same length (20μs) can carry MPDUs at different data-rates, ranging from 6 to 54Mb/s.

FC	DUR	ADDR1	ADDR2	ADDR3	SC	LLC	PAYLOAD	FCS

Fig. 6. Logical format of an IEEE 802.11 frame.

While we refer the interested readers to the original paper [8], we remark that, different to BLE, 802.11g frames are self-contained and no previous knowledge about the transmitter or the network is needed to *capture* such frames. For instance, both the binary content of the PLCP and the MPDU are scrambled and protected (with CRC codes) using polynomials with well-known coefficients. For this reason, we can easily decode frames and retrieve useful information about some of their fields, as shown in Fig. 6. These include the frame type (e.g. data, management or control), the MAC address of the sender and that of the receiver, and others (parsing is not straightforward, as it depends on the first field, i.e .the Frame Control – FC). Additionally our decoder also prints the time-stamp of the frame, anchored to its start, which we compute over the number of processed samples when the LTS part of the PLCP is decoded. The 20MS/sec sample rate allows a resolution of 50ns.

In our framework we register 13 decoders, each centered at one of the central frequencies of the corresponding 20MHz Wi-Fi channel. To receive a sequence of samples at 20MS/sec, we set $R_{\mathrm{up}} = 5$ and $R_{\mathrm{down}} = 13$. Finally, the decoders provide to the trace collector only the frames that have passed the Frame Check Sequence (FCS) test.

Preamble 1B	Access Address 4B	Protocol Data Unit (PDU) 2-257B		FCS 3B
AA or 55	8E89BED6	header (length)	Payload up to 255B	

Fig. 7. Logical format of a BLE frame.

B. BLE decoder

We developed the BLE decoder from scratch and presently this only supports the 1Mb/s Gaussian-Frequency-Shift-Keying (GFSK) encoding with modulation index in the range $[0.45 - 0.55]$. Different to Wi-Fi, capturing BLE frames on a given channel among the 40 available is not straightforward, even though the frame format used and which we show in Fig. 7 is very simple. Note however that both the Access Address (AA) and the polynomial used for protecting the frame with a 3-byte CRC are not fixed by specification.

In BLE a node can communicate either by broadcasting frames on *Advertisement* channels (37, 38 and 39), or by transmitting unicast frames to an associated peer on *Data* channels (from 0 to 36). While in the former case frames are protected with a standard CRC polynomial and addressed to a fixed *Advertisement* AA (`0x8E89BED6`), in the latter case both the AA and the CRC polynomial are negotiated by peers when they establish a communication session. To this end, one peer acts as a *peripheral* and advertises itself on the Advertisement channels. The other peer, the *central*, transmits a *Connection Request* (CR) that embeds the AA and the CRC polynomial, and *hop interval* parameters that will be used to set up the *Frequency Hopping* (FH) procedure. After the connection is established, frames are transmitted in a FH fashion. For this reason, the 40 BLE decoders that we register in our framework have to share a connection database that we fill with parameters captured from CRs that we receive on channels 37–39. Once a decoder on channels 0–36 intercepts a valid preamble, it keeps decoding the frame and checks whether the AA is in the database. If this is the case, the decoder then verifies the validity of the FCS using the corresponding polynomial. Decoders on channels 37–39 instead use the standard parameters.

In our implementation, the decoders provide the trace collector with the AA that identifies the session and the frame type extracted from the PDU header only for correct frames. This further includes the time-stamp that is anchored on the first bit of the detected preamble. Since we perform wide band capture across all the BLE channels simultaneously, we do not need to execute any FH procedure. We simply register multiple decoders centered at the central frequencies of the 40 BLE channels, setting $R_{\mathrm{up}} = 1$ and $R_{\mathrm{down}} = 26$. This ensures decoders receive a sequence of 2MS/sec, which in turn enforces a time-stamp resolution of 0.5μs.

C. *Trace synchronisation*

As we mentioned in Sec. II-A, the capture engine transmits a custom BLE *synch frame* once per second on BLE channel 18. This frame is generated with `0xBEEF` as AA and `0x123456` as CRC polynomial, which we add to the BLE database. We also register a BLE decoder on channel 18, which we feed with data coming from capture device high, while a second one is fed with data from the other device. In this way, the trace collector measures the time-stamp difference between identical frames addressed to `0xBEEF` and reported by the two BLE decoders. It then adjusts the timing of the following frames by taking into account this difference. The trace collector drops all the frames received before the first synch frame, to avoid ambiguities.

IV. Results

In this section we present the results of preliminary tests that we performed to validate our framework, while we will pursue a more comprehensive evaluation as future work. For the first test we use a pair of USRP-N210 devices, synchronised with a MIMO cable, and transmit a sequence of mixed Wi-Fi and BLE frames. To this end we tune one USRP device to 2.442GHz and the other to 2.452GHz, which are the carriers of Wi-Fi channels 7 and 9. Using a sample rate of 25MS/sec we emulate the scenario depicted in Fig. 8 where we transmit a frame every 2ms, covering BLE channels from 9 to 28. We start and end the sequence with a couple of 1,462B Wi-Fi data frames encoded at 6Mb/s, which corresponds to a transmission time of 1,982μs. In between we sweep the 20 BLE channels with 125B long frames, whose transmission takes approximately 1ms. We encode these with the AA and polynomial that we pre-stored in the database to enable decoding, except for the frame transmitted on advertisement channel 38, for which we use standard parameters. We isolate the testbed by running the capture within a concrete building basement.

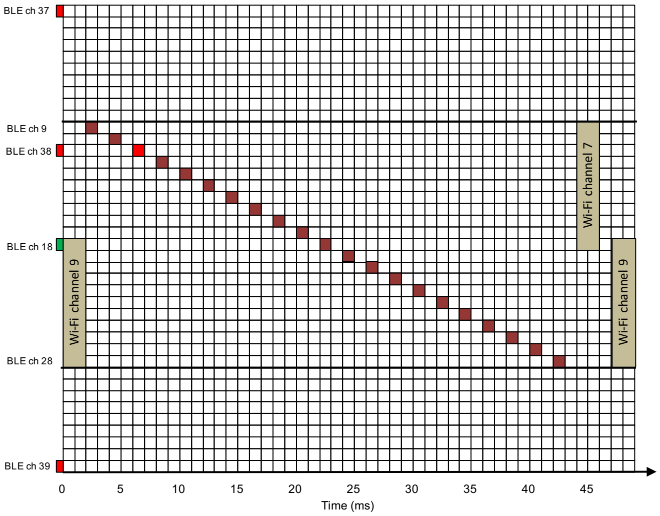

Fig. 8. Transmissions emulated for first test.

TABLE II
Output of the framework for transmissions as in Figure 8

```
WIFI:CH = 7:TS = 0:LEN = 1462B:DATA:R6:DA = ...
BLE:CH = 9:TS = 2001:LEN = 125B:AA = 12345678
BLE:CH = 10:TS = 4001:LEN = 125B:AA = 12345678
BLE:CH = 38:TS = 6000:LEN = 37B:AA = 8E89BED6
BLE:CH = 11:TS = 8000:LEN = 125B:AA = 12345678
BLE:CH = 12:TS = 10000:LEN = 125B:AA = 12345678
BLE:CH = 13:TS = 12001:LEN = 125B:AA = 12345678
BLE:CH = 14:TS = 14001:LEN = 125B:AA = 12345678
BLE:CH = 15:TS = 16000:LEN = 125B:AA = 12345678
BLE:CH = 16:TS = 18001:LEN = 125B:AA = 12345678
BLE:CH = 17:TS = 20001:LEN = 125B:AA = 12345678
BLE:CH = 18:TS = 22001:LEN = 125B:AA = 12345678
BLE:CH = 19:TS = 24001:LEN = 125B:AA = 12345678
BLE:CH = 20:TS = 26001:LEN = 125B:AA = 12345678
BLE:CH = 21:TS = 28001:LEN = 125B:AA = 12345678
BLE:CH = 22:TS = 30001:LEN = 125B:AA = 12345678
BLE:CH = 23:TS = 32000:LEN = 125B:AA = 12345678
BLE:CH = 24:TS = 34001:LEN = 125B:AA = 12345678
BLE:CH = 25:TS = 36001:LEN = 125B:AA = 12345678
BLE:CH = 26:TS = 38001:LEN = 125B:AA = 12345678
BLE:CH = 27:TS = 40000:LEN = 125B:AA = 12345678
BLE:CH = 28:TS = 42001:LEN = 125B:AA = 12345678
Wi-Fi:CH = 9:TS = 44000:LEN = 1462B:DATA:R6:DA = ...
Wi-Fi:CH = 7:TS = 46000:LEN = 1462B:DATA:R6:DA = ...
BLE:CH = 9:TS = 48001:LEN = 125B:AA = 12345678
```

We report the trace produced by our framework throughout this experiment in Table II. Observe that all the channels (CH) and the frame lengths (LEN) are correctly reported. In the case of BLE the AA value is also accurate, whilst for 802.11 frames the type (DATA) and datarate (R6 stands for 6Mb/s) extracted match the values of the transmitted frames. Note that we cut the Wi-Fi MAC addresses in this example. Finally, note the increasing time-stamps (TS) with at most 1μs uncertainty, which we expect is due to the missing synchronisation between the transmitting and receiving USRP devices that we used. The trace collector automatically sets the time-stamp of the first captured frame to zero.

For the second test we set up a BLE network with a couple of RedBear Nano v1.5 [13]. We program one acting as central peer to connect 1.7s after it receives the first advertisement from the other, which acts as peripheral. After this the central transmits data for 1s, one frame every 450ms, followed by several other data frames, almost in random order. We parse the trace output to report only the time-stamps of advertisement frames transmitted by the peripheral, other advertisements from any neighbouring nodes, and data frames transmitted by the central. We consider as $t = 0$ the time-stamp of the first advertisement transmitted by the peripheral.

Fig. 9 reports the receiving frequencies as a function of the time-stamps. Our framework detects the connection request and reports the typical frequency hopping pattern in the data frames. We also see that the peripheral stops transmitting advertisement frames after the connection has been established, while we observe that advertisements from external BLE device continue to be transmitted.

Following this preliminary results we conclude that the proposed framework for debugging IoT wireless applications can accurately discriminate between different technologies and precisely record packet timings and contents.

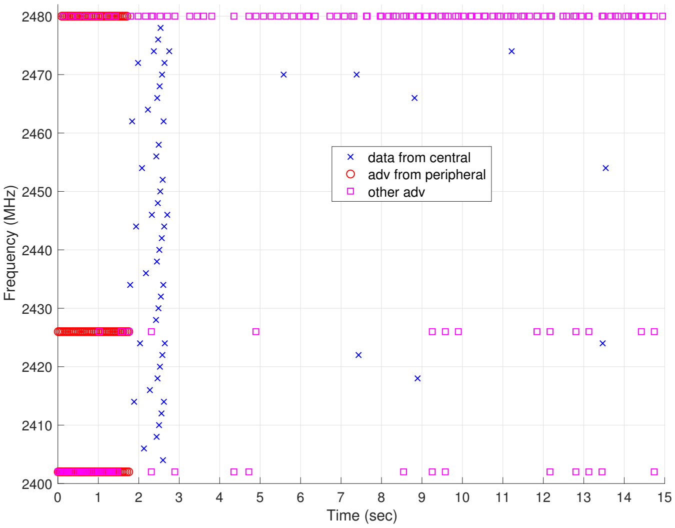

Fig. 9. Experiment with two BLE nodes. Traffic before and after the central connects to the peripheral at $t = 1.78$s. Experimental results.

V. Conclusions

In this paper we presented an extensible framework for capturing multi-standard transmissions in the 2.4GHz band. Through a set of controlled experiments, we demonstrated that the framework can correctly capture and decode frames transmitted at the same time over different frequencies by different wireless technologies (Wi-Fi and BLE), which are stored on disk for offline analysis. We have shown that the trace collector can accurately follow BLE data sessions after a proper connection request, which does not require explicit synchronisation with the hopping sequence agreed between a communicating pair. Future work will focus on extending the proposed framework to enable decoding of other Bluetooth standards including 5.0 and also of 802.11n frames, both MCS encoded and transmitted over 40MHz channels using a single spatial stream.

Acknowledgments

This work was partially funded by the projects BSL (Brescia SMART LIVING “Energia e servizi integrati per la valorizzazione del benessere”) and SCUOLA (Smart Campus as Urban Open LAbs “Smart Cities and Communities Regione Lombardia”).

References

[1] Ellisys Bluetooth Tracker, available online at https://www.ellisys.com/products/btr1/

[2] Monitor mode for QCA/ath10k wireless cards, available online at https://wireless.wiki.kernel.org/en/users/drivers/ath10k/monitor

[3] Project Ubertooth, available online at http://ubertooth.sourceforge.net/usage/start/

[4] BLUEFRUIT LE SNIFFER, available online at https://www.adafruit.com/product/2269

[5] A. K. Das, P. Pathak, C. N. Chuah, and P. Mohapatra, “Uncovering Privacy Leakage in BLE Network Traffic of Wearable Fitness Trackers”, Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications, New York (USA), Feb. 23-24, 2016.

[6] J. Classen, D. Wegemer, P. Patras, T. Spink, M. Hollick, “Anatomy of a Vulnerable Fitness Tracking System: Dissecting the Fitbit Cloud, App, and Firmware”, PACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, In Press.

[7] D. Cauquil, “BtleJuice: the Bluetooth Smart MitM Framework”, DEF CON 24, Hacking Conference, Las Vegas (US), Aug. 4-7, 2016.

[8] B. Bloessl, M. Segata, C. Sommer, and F. Dressler, “An IEEE 802.11a/g/p OFDM Receiver for GNU Radio”, Proceedings of the second workshop on Software radio implementation forum, SRIF-13, Hong Kong, China, August 12, 2013.

[9] Ettus USRP B200, available on line at https://www.ettus.com/product/details/UB200-KIT

[10] *IEEE document standard, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 2012

[11] Bluetooth Core Specifications, available on line at https://www.bluetooth.com/specifications/bluetooth-core-specification

[12] R. Coker, Bonnie++, available on line at https://www.coker.com.au/bonnie++/experimental/

[13] BLE Nano, available on line at http://redbearlab.com/blenano/

[14] E. Blossom, “GNU Radio: Tools for Exploring the Radio Frequency Spectrum”, Linux Journal, 122, June 2004.