ESPwn32: Hacking with ESP32 System-on-Chips

Romain Cayre
EURECOM
Biot, France
romain.cayre@eurecom.fr

Damien Cauquil
Quarkslab
Rennes, France
dcauquil@quarkslab.com

Aurélien Francillon
EURECOM
Biot, France
aurelien.francillon@eurecom.fr

Abstract—In this paper, we analyze the ESP32 from a wireless security perspective. We reverse engineer the hardware and software components dedicated to Bluetooth Low Energy (BLE) on the ESP32 and ANT protocol on Nordic Semiconductors' nRF chips. Exploiting this, we then implement multiple attacks on the repurposed ESP32 targeting various wireless protocols, including ones not natively supported by the chip. We make link-layer attacks on BLE (fuzzing, jamming) and cross-protocol injections, with only software modifications. We also attack proprietary protocols on commercial devices like keyboards and ANT-based sports monitoring devices. Finally, we show the ESP32 can be repurposed to interact with Zigbee or Thread devices. In summary, we show that accessing low-level, non-documented features of the ESP32 can allow, possibly compromised, devices to mount attacks across many IoT devices.

Index Terms—Internet of Things, ESP32, wireless, protocols, security, reverse engineering, cross-protocol attacks

I. Introduction

Recently, system-on-chips (SoC) are increasingly used, and provide a large set of sophisticated functionalities, such as wireless connectivity. In particular, they are largely deployed in connected objects, making their security analysis fundamental. We need to understand better these complex systems to identify and correct their vulnerabilities and explore new attack surfaces related to their deployment. The security analysis of software and hardware components involved in the wireless connectivity, in particular, remains a major challenge today, because their low level internals are often opaque and undocumented. It impacts both wireless and embedded security and mobilizes an interdisciplinary approach at the interface between electronics, signal processing, and computer science. From an embedded security perspective, the security of all the applications developed on these modules depends on a large set of complex software components, interconnected and partially documented. From a wireless security perspective, the analysis of these components is also fundamental, as they implement increasingly complex protocol specifications with a large set of features and real-time constraints. Understanding, analyzing, and instrumenting their internals both facilitate the analysis of wireless protocols but can also lead to new attack vectors.

In this article, we present our analysis of the internals of the Bluetooth Low Energy [5] stack embedded on ESP32, ESP32-S3, and ESP32-C3 SoCs from Espressif Systems. These series of SoCs are increasingly used today, as they provide a very large set of features, including Wi-Fi, Bluetooth BR/EDR, and Bluetooth Low Energy connectivity, for a very low cost. It makes them especially attractive for developing connected devices, and they are widely used by *makers* and *hobbyists*, but also in some commercial products, including industrial systems [25], [41]. We present an extensive overview of these SoCs, from their global architecture to the software and hardware components involved in BLE connectivity. We focus on the reverse engineering process of the lowest layers of the protocol stack, and we describe how the software and hardware components can be diverted to implement a wide set of advanced offensive techniques.

We demonstrate that these low-level components can be diverted from software to gain fine-grained control over the Link Layer and the Physical Layer of the stack, allowing to build powerful primitives from an offensive perspective without any hardware modifications. We show the feasibility of injecting, monitoring, and altering Link Layer traffic on the fly. We also explore how the physical layer can be diverted at different levels, from the modulators and demodulators configuration to the radio calibration process. These techniques allowed us to attack the BLE protocol itself but also various other wireless protocols, such as ZigBee [42], MosArt [26], ANT+ [23] or Riitek. These protocols are not natively supported by the ESP32 SoCs but coexist in the same frequency band and share some similarities in terms of modulation, allowing us to interact with them using cross-protocol techniques. Finally, we highlight serious security flaws in the design of several proprietary protocols and exploit them to perform wireless attacks targeting sensitive devices, such as wireless keyboards or heart rate monitors.

In summary, we tackle the following challenges:

- We provide a fine-grained analysis of the proprietary hardware and software components related to BLE communications in ESP32 SoCs,
- We provide an instrumentation approach allowing to manipulate the packet flow,
- We demonstrate how the radio components can be repurposed to attack non-native protocols,
- We reveal and exploit severe weaknesses in the design of ANT and Riitek proprietary protocols.

II. RELATED WORK

In recent years, there has been a particular interest in the analysis and hacking of SoCs, especially in wireless security. Indeed, a close relationship exists between SoCs hacking and the discovery of new offensive techniques. Numerous hacks have been carried out in the context of applied research on the security of wireless protocols and have allowed the exploration of new attack strategies. Furthermore, these works often push chips to the limits of their technical capabilities by exploiting hardware architecture details or vulnerable software implementations, thus opening new technical and scientific perspectives.

These works often aim to solve concrete technical issues or limitations. Indeed, the security analysis of wireless protocols remains complex today and regularly confronts the limits of existing analysis tools. For example, Software Defined Radios (SDRs) offer genericity but remain expensive, require consequent engineering work, and impose limits on their reduced bandwidth and the latency associated with their design. In particular, protocols involving channel hopping algorithms require real-time monitoring of the channel sequence or wide band monitoring, requiring very fast or expensive tooling. These limitations motivated the implementation of dedicated hardware for Bluetooth BR/EDR and BLE [31]. Other examples include hardware tools developed as part of the analysis of proprietary wireless keyboard protocols, such as KeyKeriki [29], or the analysis of protocols based on the IEEE 802.15.4 specification [22] (e.g., Thread, ZigBee, Wireless Hart) through the APImote [18].

System-on-Chip hacks have complemented these dedicated tools, significantly decreasing the cost of analysis tools and enabling new attack strategies. For example, the repurposing of a hardware register on Nordic Semiconductor's nRF24L01 chip by Travis Goodspeed to allow passive eavesdropping on proprietary protocols in the 2.4 GHz band [20] was a decisive step, allowing the development of a dedicated analysis firmware [27] and the discovery of multiple vulnerabilities targeting various wireless keyboards and mice [26]. Many works have extended this hack to nRF51 and nRF52 chips, allowing the development of offensive tools on various platforms [7], [8], [13] and the discovery of critical low-level attacks targeting the BLE protocol [9], [11]. Another example is modification of ATMEL's RZUSBStick firmware by Joshua Wright to provide injection capabilities as part of his work on the security of the ZigBee [40] protocol.

Other hacks have focused on reverse engineering and diverting existing embedded protocol stacks. Mathy Vanhoef and his tool *modwifi* allowed the development of low-level attacks targeting the Wi-Fi [38] protocol. Similarly, the work of Matthias Schulz et al. explored the hacking of Broadcom proprietary Wi-Fi stacks [30]. The *InternalBlue* framework [24], dedicated to Bluetooth and Bluetooth Low Energy, allowed the analysis and instrumentation of Broadcom and Cypress proprietary protocol stacks. This work has been a strong basis for multiple offensive works [2], [3], [17]. We can
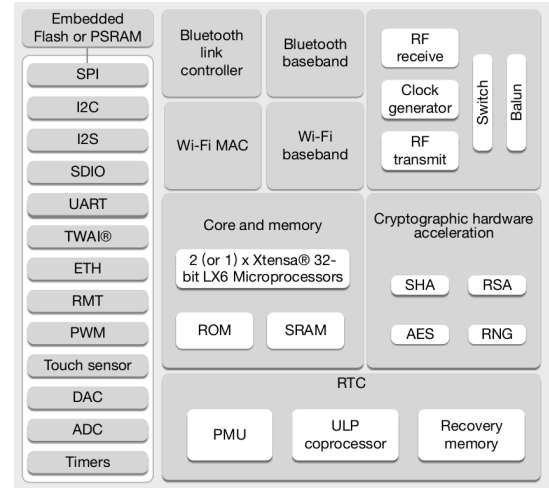

Fig. 1. ESP32-D0WDQ6 architecture (ESP32 series), from ESP32 datasheet [35], (version 4.2, section 1.6, figure 1, page 12).

also mention Matheus Garbelini and his Bluetooth Classic OTA *fuzzer* based on ESP32, which initiated the reverse engineering and modification of some parts of its Bluetooth Classic stack. This work leads to the discovery of a serie of vulnerabilities named *BrakTooth* [16]. Let us note that ESP32 SoCs in particular are very common in the IoT context, and often used as an experimental platform in academic research (e.g., RealSWATT [32], where ESP32 is used to implement a prototype of a remote software-based attestation for embedded devices).

Our work is a continuation of these SoC hacks by expanding the reverse engineering of the ESP32 platform from an offensive perspective and extending it to its ESP32-S3 and ESP32-C3 variants. In particular, we focus on the analysis and hacking of the proprietary embedded Bluetooth Low Energy protocol stack and the hardware components involved in its operation.

III. ESP32 HARDWARE ARCHITECTURE

A. Global Architecture

ESP32 systems integrate one or more processors, a radio communication layer, one or more protocol stacks (Wi-Fi, Bluetooth Low Energy, Bluetooth BR/EDR), a cryptographic accelerator, and various peripherals allowing them to interface with a large number of devices: displays, SD card, Ethernet interface, etc. Figures 1 shows an example of a popular system architecture, *ESP32-D0WDQ6* (used in the development boards *ESP32-WROOM-32*).

B. Instruction Set Architectures

ESP32 systems are currently based on two main Instruction Set Architectures (ISA): XTensa from Tensilica (used in ESP32 and ESP32-S series) and RISC-V (used in ESP32-C, ESP32-H, and ESP32-P series). Espressif Systems recently announced that future series will only use RISC-V ISA [15].

C. Memory layout

The main component of ESP32 is the *Core System*. It includes the processors, static Random Access Memory (*SRAM*), and Read Only Memory (*Mask ROM*). An embedded or external flash memory is also used to store the user application's code and data.

According to the memory maps detailed in the datasheets [33]–[35], two memory regions are associated with Mask ROM. They are respectively mapped at `0x40000000` and `0x3FF90000` in the case of ESP32-D0WDQ6, or `0x40000000` and `0x3FF00000` in the case of ESP32-C3 and ESP32-S3 series. These regions mainly store the low-level software components of Wi-Fi, Bluetooth BR/EDR and Bluetooth Low Energy protocol stacks, and various low-level functions. They are fully integrated into the SoC and can't be modified.

The static RAM is divided into three main regions, respectively named SRAM0, SRAM1, and SRAM2. The internal RAM layout is split according to the Harvard architecture. An Instruction Memory (IRAM) is used to store timing critical code, and a Data Memory (DRAM) is used to store various data segments. The IRAM is typically mapped in SRAM0, while the DRAM is mapped in SRAM1 and SRAM2, which are contiguous in memory. Similarly, the flash is also segmented into a Data Memory (DROM), used to store constant data, and an Instruction Memory (IROM), containing the code of the user application and libraries provided by Espressif Systems.

D. Wireless components

ESP32 series includes two main components dedicated to wireless communications: the radio module and the wireless MAC and baseband. The radio module provides the analog frontend, including a 2.4GHz RF transmitter, a 2.4GHz RF receiver, and a clock generator. Its use is shared between the wireless MAC and baseband of Wi-Fi and Bluetooth protocols, which are in charge of low-level operations, such as the modulation and demodulation of baseband signals, packet flow processing, and access control. Espressif Systems provides few details about these components' behavior and interactions with the *Core System*. In the specific case of Bluetooth baseband, the modulator and demodulator rely on a frequency modulation named Gaussian Frequency Shift Keying (GFSK) with a data rate of 1Mbps or 2Mbps, depending on the physical layer in use.

IV. BLUETOOTH LOW ENERGY STACK REVERSE ENGINEERING

A. Overall methodology

Our main objective was understanding how the Bluetooth Low Energy stack is implemented on these SoCs, especially how the low-level operations are managed. Indeed, a typical BLE protocol stack is split into two main components: the Host, which is in charge of the highest layers, providing security and applicative features, and the Controller, handling the lowest layers, e.g., the Physical and the Link Layer.

These two parts typically communicate using a standardized interface named Host Controller Interface (HCI). Identifying the specificities of the implementation of this stack is a mandatory step to explore how they can potentially be diverted to implement low-level attacks.

We combined a static analysis, mostly performed using the Ghidra tool, with a dynamic analysis, by compiling example codes related to Bluetooth Low Energy provided in the Software Development Kits (SDKs) and modifying them to validate our assumptions or gain a better knowledge of a given functionality.

B. Overcoming static analysis challenges

Several issues had to be overcome to analyze the BLE stack statically.

1) Analyzing XTensa instructions: First, while the ESP32-C3 series are based on RISC-V ISA, based on an open standard and natively supported by Ghidra, it is not the case for ESP32 and ESP32-S3 series, which are based on XTensa ISA. Ghidra does not natively support this ISA, we used a Ghidra extension developed by the community [14] to add partial support for XTensa ISA. Thanks to this extension, we could compile code examples, load the generated ELF files into Ghidra and analyze them.

2) Extracting and exploiting the Mask ROM regions: The second issue was linked to the fact that a significant part of the stack, especially the low-level functions, are implemented in the Mask ROM regions and are not included in the ELF symbols. Hopefully, Espressif Systems provides a python tool as part of its development environment [36], named *esptool.py* [37], which allows extracting the content of Mask ROM regions.

The extracted ROM memory regions can't be used directly. Although we know its memory location, we must identify the functions and their addresses in the ROM. Fortunately, the development environment relies on linker scripts containing those functions' addresses. A code snippet extracted from one of these scripts is presented in listing 1.

```
/*
ESP32 ROM address table
Generated for ROM with MD5sum:
ab8282ae908fe9e7a63fb2a4ac2df013  ../../rom_image/prorom.elf
*/
PROVIDE ( Add2SelfBigHex256 = 0x40015b7c );
PROVIDE ( AddBigHex256 = 0x40015b28 );
PROVIDE ( AddBigHexModP256 = 0x40015c98 );
PROVIDE ( AddP256 = 0x40015c74 );
PROVIDE ( AddPdiv2_256 = 0x40015ce0 );
PROVIDE ( app_gpio_arg = 0x3ffe003c );
PROVIDE ( app_gpio_handler = 0x3ffe0040 );
PROVIDE ( BasePoint_x_256 = 0x3ff97488 );
PROVIDE ( BasePoint_y_256 = 0x3ff97468 );
PROVIDE ( bigHexInversion256 = 0x400168f0 );
PROVIDE ( bigHexP256 = 0x3ff973bc );
PROVIDE ( btdm_r_ble_bt_handler_tab_p_get = 0x40019b0c );
```

Listing 1: Code snippet extracted from linker scripts.

The ESP32 ROM segments are then loaded into memory and placed at the correct addresses in Ghidra, and the symbols related to them (extracted from the linker file) are added using a custom script. It is then possible to decompile the ROM
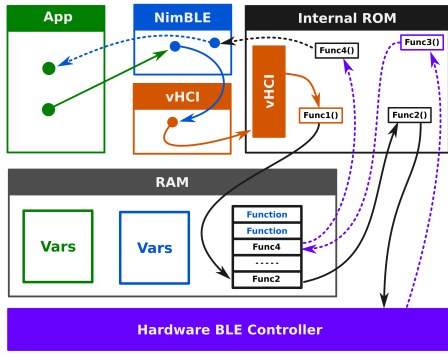
Fig. 2. Bluetooth Low Energy ESP32 architecture.

code with function names and determine its interaction with the application code.

C. BLE protocol stack analysis

Espressif Systems allows using two popular open-source BLE stacks, NimBLE [4] and BlueDroid [1]. However, these components only implement the Host layers: most of the BLE protocol operations related to the low layers are implemented in the functions stored in Mask ROM and interfaced with NimBLE or BlueDroid using a virtual Host Controller Interface (vHCI). As a result, these components can't directly manipulate the PDU exchanged by the BLE protocol.

The low-level functions stored in Mask ROM are generally called indirectly, using a set of arrays containing function pointers stored in RAM and initialized at startup. This mechanism allows Espressif Systems to patch these functions by embedding a new implementation in IRAM and redirecting the control flow by replacing the function pointers. Figure 2 summarizes the architecture and the interconnections between the application, the vHCI adapter embedded in the ROM, and the Bluetooth Low Energy hardware controller embedded in the SoCs.

This BLE hardware controller is not documented, but we identified it by analyzing various registers' addresses and finding a *pastebin* referencing the DA14681 [28] component. It has registers similar to those used on the ESP32, and we deduced that it was very likely that Espressif had integrated an existing controller but placed it at a different address (`0x3FF71200`). We could reproduce similar observations on ESP32-C3 and ESP32-S3 series, mapped at another address (`0x60031000`).

We were thus able to observe in the code of the function *r_lld_adv_start* instructions configuring the registers `BLE_ADVCHMAP_REG` and `BLE_ADVTIM_REG` managing respectively the configuration of the advertising channels and the advertising interval, as shown in the figure 3.

We identified a shared memory area used to program and exchange data with the hardware controller by analyzing several low-level functions. This memory area, named *Exchange Memory*, is used to store multiple structures specific to the hardware controller, *descriptors*, which are intended to store

```
_BLE_ADVCHMAP_REG = (uint)*(byte *)(iVar3 + 0x16);
memw();
if (0x1f < *(byte *)(iVar3 + 0x19)) {
  iVar5 = (*(byte *)(iVar3 + 0x19) + 0x10) * 8 + 0x564;
}
memw();
_BLE_ADVTIM_REG = iVar5;
```

Fig. 3. Configuration of advertising intervals and channels in *r_lld_adv_start()*.

the metadata linked to BLE PDUs (channel on which a PDU was received, the signal level, and the type of PDU) and the associated transmission and reception buffers. The protocol stack uses it to indicate to the BLE hardware controller that it has PDUs to send and also by the controller to transmit to the protocol stack the received PDUs.

V. INSTRUMENTING AND HACKING THE STACK

Based on this analysis, we explored the feasibility of hacking the stack to achieve multiple objectives:

- intercept, modify, and inject arbitrary PDUs in a BLE communication,
- divert the hardware controller to interact with non-natively supported protocols,
- divert the radio module to jam communications or build a covert channel.

To achieve these objectives, we need to hijack the control flow of low-level functions, manipulate the memory to alter data structures and acquire direct control over the hardware controller.

A. Low-level functions hooking

In subsection IV-C, we highlighted that low-level functions are mainly called indirectly, using arrays of function pointers. Some of these function pointers are dedicated to specific events and act as callbacks. We can use this mechanism to redirect the control flow to our own functions when a specific event occurs, such as the reception of a BLE PDU, by replacing the corresponding function pointer in the associated array. The memory access is not protected, allowing us to set up hooks at various places of the protocol stack. When a call occurs, we can execute arbitrary code and transparently redirect the control flow to the original function. This hooking technique is illustrated in Figure 4.

B. Manipulating BLE communications

1) Interception of BLE PDUs: Thanks to this hooking technique, we could set up a hook on the main function processing received PDUs. The regular behavior of this function relies on access to the exchange memory, which contains the received PDUs and their associated metadata. The hook also has read and write access to this memory region. As a result, we can:

- *monitor the received BLE PDUs* by reading the corresponding buffers,
- *prevent the stack from processing a PDU* by altering the PDU header to set its size field to zero (empty PDU are commonly exchanged during a BLE connection),
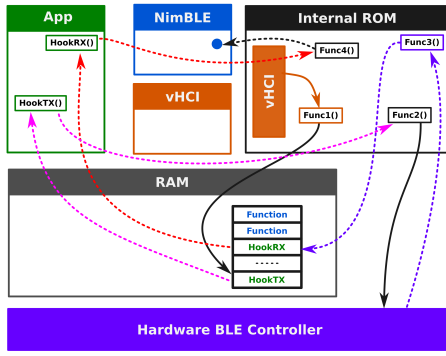
Fig. 4. Hooking of reception and transmission functions.

- *forcing a specific stack behavior* by altering the PDU itself.

We also install a hook on the function called during a PDU transmission, allowing it to intercept PDU before their transmission to the hardware controller. As a result, we can modify or even block them on the fly. Listing 2 presents one of the hooks developed during this research.

```
/**
 * _lld_pdu_data_send()
 *
 * This hook is called when the BLE stack sends a data PDU.
 **/

int _lld_pdu_data_send(struct hci_acl_data_tx *param)
{
  struct em_buf_tx_desc *p_desc = NULL;
  uint8_t *ptr_data;
  int i, forward=HOOK_FORWARD;
  struct co_list_hdr *tx_desc;
  struct em_desc_node *tx_node;

  if (gpfn_on_tx_data_pdu != NULL)
  {
    /* Should we block this data PDU ? */
    if (gpfn_on_tx_data_pdu(
        0,
        (uint8_t *)(p_rx_buffer + param->buf->buf_ptr),
        param->length
    ) == HOOK_BLOCK)
    {
      /* Set TX buffer length to zero (won't be transmitted,
         but will be freed later. */
      param->length = 0;
    }
  }

  /* Forward to the original function. */
  return pfn_lld_pdu_data_send(param);
}
```

Listing 2: Hook allowing to alter a PDU transmission.

2) Injection of arbitrary PDUs: We can inject an arbitrary PDU into a BLE connection by exploiting a vulnerability in the `lld_pdu_data_tx_push` function. Indeed, this function is normally called to push data PDU (linked to the L2CAP layer) into a chained list of TX descriptors ready for transmission by the hardware controller. However, as it does not check the LLID field of BLE packets, we can divert it to transmit data and control PDUs. We can allocate a transmission buffer and the associated descriptor in the exchange memory, write our arbitrary PDU into it and call the function. These operations

are time critical and directly manipulate the exchange memory, which requires implementing them in the IRAM. This prevents latency due to the flash access and requires to temporarily disable interrupts. The implementation allowing this injection is presented in listing 3.

```
void IRAM_ATTR send_raw_data_pdu(int conhdl, uint8_t llid,
                                 void *p_pdu,int length,
                                 bool can_be_freed)
{
  struct em_buf_node* node;
  struct em_desc_node *data_send;
  struct lld_evt_tag *env = (struct lld_evt_tag *)(
    *(uint32_t*)((uint32_t)llc_env[conhdl]+0x10) + 0x28
  );

  portDISABLE_INTERRUPTS();
  /* Allocate data_send. */
  data_send = (struct em_desc_node *)em_buf_tx_desc_alloc();

  /* Allocate a buffer. */
  node = em_buf_tx_alloc();

  /* Write data into the allocated buf node. */
  memcpy(
    (uint8_t *)((uint8_t *)p_rx_buffer + node->buf_ptr),
    p_pdu, length
  );

  /* Write information into our em_desc_node structure. */
  data_send->llid = llid;
  data_send->length = length;
  data_send->buffer_idx = node->idx;
  data_send->buffer_ptr = node->buf_ptr;

  /* Call lld_pdu_data_tx_push */
  pfn_lld_pdu_data_tx_push(env, data_send, can_be_freed);

  env->tx_prog.maxcnt--;

  portENABLE_INTERRUPTS();

}
```

Listing 3: Implementation of arbitrary PDU injection.

3) Evaluation: We evaluated our BLE PDUs manipulation primitives by establishing BLE connections with various devices and reproducing manually multiple BLE procedures. We could perform these procedures reliably, from the discovery process of applicative layers (e.g., GATT service and characteristics) involving data PDUs to the version discovery procedure, based on a specific type of control PDU named `LL_VERSION_IND`.

This last procedure is especially interesting from an offensive perspective because it allows retrieving the Bluetooth Low Energy baseband vendor and software version. We can turn it into a fingerprinting approach, allowing us to identify the protocol stack used by a given device accurately. We implemented a prototype on an ESP32-based smartwatch, T-Watch 2020 from Lilygo, as illustrated by Figure 5. We can then check if some vulnerabilities have been published and inject the traffic allowing to trigger them, including malformed ones. This operation could be automated, leading to a powerful penetration testing tool that can compromise surrounding devices automatically.

C. Interacting with non-natively supported protocols

We also explored the feasibility of diverting the hardware controller to implement cross-protocol attacks. These
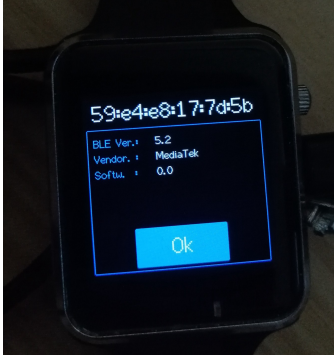
Fig. 5. Lilygo T-Watch running our custom firmware.

```
09:02 // offset: 0 - CNTL
00:08
05:10 // offset: 4 - THRCNTL_RATECNTL
6a:5d
d6:a1
df:7c
be:d6 // offset: 12 - SYNCL
8e:89 // offset: 14 - SYNCW
55:55 // offset: 16 - CRCINIT0
55:00 // offset: 18 - CRCINIT1
00:00 // offset: 20 - FILTPOL_RALCNTL
25:00 // offset: 22 - HOPCNTL
0a:00 // offset: 24 - TXRXCNTL
30:80 // offset: 26 - RXWINCNTL
00:14 // offset: 28 - TXDESCPTR
00:00
30:00
00:00 // offset: 34 - LLCHMAP0
00:00 // offset: 36 - LLCHMAP1
00:00 // offset: 38 - CHMAP2
00:00 // offset: 40 - RXMAXBUF
```

Listing 4: Annotated dump of a control structure, extracted from ESP32-C3 exchange memory.

attacks aim to interact with wireless protocols which are not natively supported by the transceiver but co-exist in the 2.4GHz frequency band and share some similarities in terms of modulation. Previous works [6], [12] have partially explored this research direction and highlighted the potential attack surface opened by such attacks.

1) Building arbitrary GFSK reception and transmission primitives: Implementing this kind of attack requires very low-level control over the operations implemented in the BLE hardware controller. In particular, we need to acquire direct or indirect control over multiple operations performed while processing received and transmitted packets, such as the whitening process, the integrity checking mechanism, the frequency configuration, or the data rate. Altering these operations allows diverting the hardware controller to build generic reception and transmission primitives based on GFSK modulation. A simplified representation of the reception primitive is presented in Figure 6.

To implement such primitives on ESP32 SoCs, we diverted the *scanning* and *advertising* modes of BLE protocol. These modes receive and transmit *advertising* packets and are good candidates to implement our cross-protocol primitives. Indeed, they are less complex than the *connected* mode and do not require establishing a BLE connection. Using an approach similar to the one described in Subsection V-B, we set up hooks on the functions related to the reception and transmission of *advertising* PDUs. We also hooked some configuration functions, called at the initialization of the mode, and dedicated to the hardware controller configuration. Thanks to these hooks, we can alter the configuration parameters on the fly by modifying control structures in the exchange memory.

Control structures contain a series of 16-bit registers, allowing to program various low-level operations performed by the hardware controller. We could identify and divert the role of several of these registers. An annotated dump of a typical control structure is presented in listing 4.

First, we configured the CNTL register to force the packet format to *Test Mode*. According to BLE specification [5], this mode is used for RF testing and allows the hardware controller to remove multiple checks automatically performed on the packet format.

We also modified SYNCL and SYNCW registers. These registers are used to configure the *access address*, a 32-bit long identifier at the beginning of BLE packets. As the hardware controller uses it to identify the start of a BLE packet in the demodulator output bit-stream, we can use it as a synchronization word to detect a specific pattern. By configuring these registers, we can synchronize on arbitrary preambles, including the ones used by wireless protocols which are not natively supported but relies on a similar physical layer. We also need to extend the maximum size of the reception buffer by altering the RXMAXBUF register, allowing us to extract packets up to 255 bytes.

Controlling the central frequency implies disabling the channel hopping mechanism by altering the HOPCNTL register. This register can also be used to select the BLE channel. However, using this mechanism on its own to choose the central frequency limits our primitives to the frequencies overlapping BLE channels. We can circumvent this limitation by altering another part of the exchange memory. Indeed, we identified that a specific area contains an array that maps each BLE channel to its central frequency by indicating an offset (in MHz) from 2402MHz. We can use this mechanism by configuring an arbitrary channel and remapping its associated central frequency to the one we want to use.

The data rate can be configured on ESP32-S3 and ESP32-C3 series using THRCNTL_RATECNTL. Indeed, these series support both *LE 1M* and *LE 2M* physical layers, allowing the selection of a data rate of 1Mbps or 2Mbps. Let us note that this register is not present on the ESP32 series because it does not support the *LE 2M* physical layer.

Finally, we need to disable both CRC checking and whitening. This is a key aspect of our reception primitive because we want to receive arbitrary packets that won't be valid BLE frames. It can be done by manipulating two specific bits in a global register, BLE_RWBLECNTL_REG, mapped at 0x3FF71200 for ESP32 series and 0x60031000 for ESP32-S3 and ESP32-C3 series. This register also contains a SYNCERR field, indicating the maximum number of bit-flips the demodu-
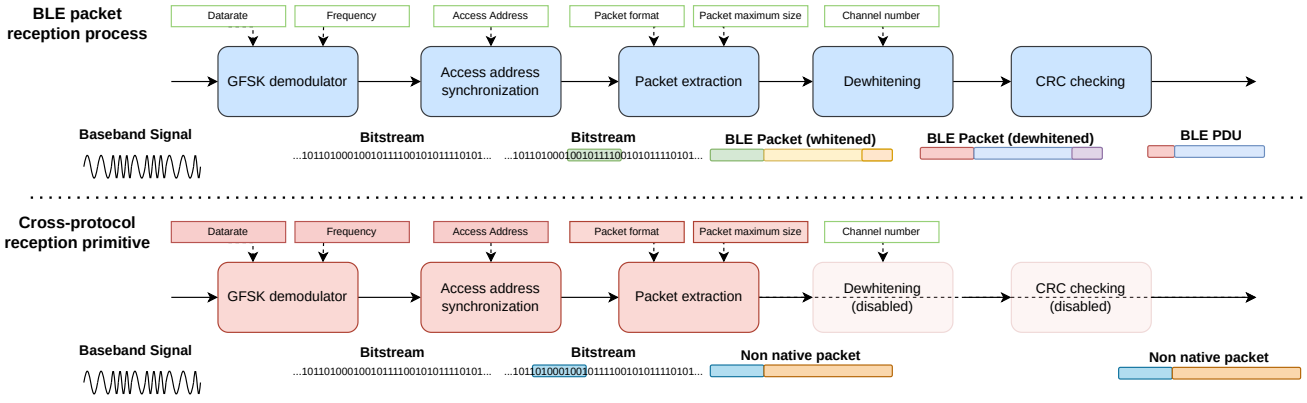
Fig. 6. Simplified representation of the cross-protocol reception primitive.

lator will tolerate in the synchronization word. We can use this feature to increase the probability of receiving packets from other protocols.

Once this configuration step has been done, we can use an implementation similar to the one described in Subsection V-B to receive and transmit arbitrary GFSK-modulated data. We obviously need to prevent the stack from processing received packets, as they are not valid BLE packets and will probably trigger a crash. On the transmission side, we can use a packet-in-packet [21] strategy to control the modulated data fully.

2) Attacking ANT protocol:

a) Protocol overview: ANT protocol is a proprietary protocol designed by Dynastream Innovations Inc., a division of Garmin Canada Inc. It operates in the 2.4GHz ISM band and is mainly used by sports-oriented devices, such as Heart Rate Monitors or smartwatches. It can be used autonomously, but two main variants are generally deployed in the wild:

- **ANT+ protocol:** used to transmit a small amount of data regularly, a specialization of ANT protocol for interoperability purposes. It establishes communication according to a Master/Slave topology between sensor devices (such as a Heart Rate Monitor) or devices intended to display or process this information (such as a smartwatch). Due to its wide deployment in sports and health-oriented devices, it transmits potentially sensitive health-related data.
- **ANT-FS (File Sharing) protocol:** used for the transmission of files between a device playing the role of a file server (called *Client*) and a client (called *Host*). It is generally used to transfer reports and histories between sports-oriented devices and a USB dongle, but also for updating the firmware (OTA) of devices (e.g., smartwatches), making it critical from a security perspective.

b) Packet format: The manufacturer provides documentation of the high layers of the protocol stack, but the low layers are not documented. Therefore, we used reverse engineering to determine the packet format and the physical layer. We performed a black-box analysis of radio communications between various devices. We also statically analyzed the ANT protocol stack implementation integrated on the nRF52 chip
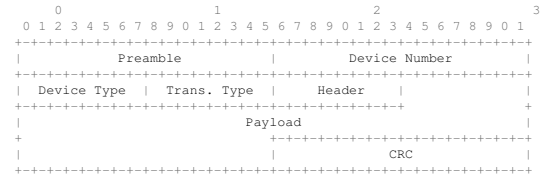
Listing 5: ANT Packet Format.

(so-called *ANT SoftDevice*). We determined that the ANT protocol is based on a 1Mbps GFSK modulation, similar to the LE 1M physical layer of Bluetooth Low Energy. The packets have a fixed size of 16 bytes, their format is presented in Listing 5. Packets are not encrypted nor authenticated by default. *Device Number* field is a unique device identifier and can be assimilated to an address. *Device type* indicates the type of devices according to a set of predefined profiles (e.g., Heart Rate Monitors, Speed Cadence, Blood Pressure) and defines the payload format accordingly. *Transmission Type* defines some characteristics linked to the communication channel. The *Header* includes several fields describing the current packet, such as the diffusion mode in *broadcast* or *unicast* or if the packet is an acknowledgment. CRC is computed using CRC-16 CCITT.

c) From Network Keys to preambles: ANT website and specification [23] describes a feature named *Network Key*, uniquely identifying a network and providing a measure of security and access control. While ANT+ and ANT-FS keys are public, the company charges a fee on demand to generate a private network key.

A *Network Key* is a sequence of 8 bytes. We observed a correlation between this *Network Key* and the 2-byte long preamble value by analyzing ANT+ and ANT-FS communications over the air. The reverse engineering of the *ANT SoftDevice* confirmed this assumption. We identified a function in charge of validating the key and generating the preamble using a simple XOR-based algorithm with predefined values.

Fig. 7. Experimental setup for malicious ANT+ packet injection.

A similar implementation is reproduced in appendix B[1]. As a result, the *Network Key* mechanism cannot be considered a security measure. A custom preamble does not guarantee access control, making ANT communications broken by design.[2]

d) Attacking ANT+ with cross-protocol primitives: In the specific case of ANT+, the generated preamble is `0xa6c5`, and only one communication channel at 2457MHz is systematically used. We implemented a helper function allowing us to switch from Little Endian (used by BLE) to Big Endian (used by ANT). We can match ANT GFSK modulation by configuring the data rate to 1Mbps. As our reception primitive requires a 4-byte long synchronization word, we can apply a two-step strategy to synchronize on an ANT+ communication:

- **Scanning:** we configure the frequency at 2457MHz and use `0xaaaaa6c5` as synchronization word. It allows matching a subset of ANT+ packets, which can then be used to extract the *Device Numbers*.
- **Sniffing:** we modify the synchronization word to match a specific communication by concatenating the `0xa6c5` preamble value with the targeted *Device Number*.

e) Evaluation: We implemented these cross-protocol primitives on the ESP32, ESP32-S3, and ESP32-C3 series. We evaluated them by eavesdropping on ANT+ communications between two models of Garmin smartwatches (Garmin Forerunner 45 and Garmin Instinct 2) and a Heart Rate Monitor. We extracted the heart rate from the packet flow in real time. We also injected malicious packets into ANT+ communication, as illustrated in Figure 7.

3) Attacking wireless keyboards proprietary protocols: Previous security-related works [13], [26], [29] have highlighted the massive use of insecure proprietary protocols operating in the 2.4GHz ISM band for wireless keyboards and mice. Most of them are based on GFSK modulation using 1Mbps or 2Mbps data rate, making them potentially compatible with our

cross-protocol primitives. We tested this assumption on two of them, MosArt and Riitek.

a) MosArt protocol: MosArt is one of the vulnerable proprietary protocols analyzed by Marc Newlin in the context of his research work on wireless keyboards and mice [26]. Based on a 1Mbps GFSK transceiver developed by MosArt Semiconductors, this protocol is massively deployed in the wild and used by many manufacturers (e.g., EagleTek, Anger, Advance, itWorks), despite its intrinsically insecure design. Indeed, it does not provide any security mechanisms and mainly relies on security by obscurity. A typical frame format is composed of a preamble (`0xAAAA`), a 4-byte long address, a basic header, a payload, a CRC (CRC16-XMODEM), and a constant postamble (0xFF). A basic XOR-based scrambling algorithm is also applied. We implemented a set of helper functions to deal with the scrambling and the endianness. We also implemented MosArt reception and transmission primitives using a synchronization strategy similar to the one used for ANT+. We observed some bit flips in the reception primitive, which may be linked to a small frequency offset or a different modulation deviation. Despite this bit flipping, we implemented a reliable wireless key-logger to read and decode keystrokes transmitted between a *ItWorks* wireless keyboard and its dongle. We also managed to inject arbitrary keystrokes into the dongle.

b) Riitek protocol: We also reverse-engineered the proprietary protocol used by Rii i8 Wireless Mini Keyboard & Touchpad to communicate with its dongle and identified serious flaws in the protocol design.[3] This protocol, designed by Riitek, is also based on a 1Mbps GFSK modulation, making it compatible with our reception and transmission primitives. It communicates using a single channel at 2426MHz and provides no encryption or authentication mechanism. HID scan codes are transmitted in plaintext over the air, making it trivial to perform both eavesdropping and malicious packet injection. A more detailed overview of the protocol can be found in appendix C. We managed to implement reliable reception and transmission primitives on ESP32, ESP32-C3, and ESP32-S3 series and were able to implement a wireless key-logger and inject keystrokes and mouse movements into the communication.

c) Generalization to other proprietary protocols: Multiple works [13], [26], [29] have highlighted serious security flaws in other wireless keyboards' proprietary protocols, such as Logitech Unifying or Microsoft. Further investigations must be conducted to evaluate the feasibility of implementing compatible cross-protocol reception and transmission primitives. Enhanced ShockBurst seems widely used to implement the lowest layers of these proprietary protocols and would probably be compatible with our primitives since it relies on 1 or 2Mbps GFSK modulation. Some specific technical challenges, such as channel hopping algorithms or physical layer differences (e.g., frequency deviation, data rate), could

---

[1]The reverse engineering of *ANT SoftDevice* has been performed in compliance with local laws and regulations. The code reproduced in the appendix was written by a third person based on a specification provided in appendix A to prevent copyright infringement.

[2]These security issues have been responsively disclosed to Garmin, who decided to update the ANT website to better reflect the current state of ANT security.

[3]The reverse-engineering process was based on a black-box approach based on RF communications monitoring. The vendor was notified of those security issues in June 2021. As far as we know, no fixes have been released.

limit the applicability of our approach depending on the targeted protocol and the support of the 2Mbps data rate by the diverted stack.

4) Attacking 802.15.4-based protocols: Several major protocols of the Internet of Things are operating in the 2.4GHz ISM band, such as Zigbee [42] or Thread, rely on IEEE 802.15.4 specification [22] for their lower layers. They rely on an *Offset-Quadrature Phase Shift Keying* (O-QPSK) modulation, with 16 communication channels. We implemented reception and transmission primitives for 802.15.4 protocols by reusing *WazaBee* [12] attack. Indeed, this approach exploits similarities between GFSK modulation used by BLE and O-QPSK used by 802.15.4. Thus, it is possible to build an equivalence table between the symbols used by these two modulations and implement a set of conversion functions allowing one to switch from one to the other easily. We could implement a proof of concept on ESP32-S3 and ESP32-C3 series by sniffing and injecting valid ZigBee packets. *WazaBee* attack relying on *LE 2M* physical layer, it is not possible to implement it on ESP32 series, as they only support *LE 1M* with 1Mbps data rate.

D. Diverting the radio module

During our analysis, we noticed the presence of several low-level functions in *Mask ROM*, dedicated to the configuration of the radio module, providing RF frontend to Wi-Fi, Bluetooth BR/EDR, and Bluetooth Low Energy. We also explored the feasibility of exploiting these functions to divert the radio module itself.

1) Radio module calibration: The radio module implements a typical transceiver architecture based on I/Q up-and downconversion. This architecture is known to be vulnerable to I and Q branch mismatch in the analog front-end, a so-called I/Q imbalance, which can be corrected using digital calibrations techniques.

According to Espressif IoT Development Framework documentation [36], RF calibrations algorithms are implemented on ESP32 SoCs. While Espressif does not provide low-level details on the calibration techniques, the documentation mentions that a full or partial calibration must be done before using the radio module. When a full calibration is performed, the calibration-related data are stored in *Non-Volatile Storage* (NVS). It allows reusing them subsequently during partial calibration to reduce calibration costs at startup.

2) Forcing a full calibration: A full calibration process is automatically performed if the calibration data is missing or unavailable. We can use this behavior to force a full calibration at any time. It can be done by disabling the radio module, flushing calibration data in NVS, and re-enabling the radio. This process can be performed by calling various functions the SDK exposes, as shown in listing 6.

3) Diverting the calibration process: Similarly to BLE low-level functions, calibration-related functions are stored in *Mask ROM* and called indirectly using an array of function pointers. A reference to this array can be easily accessed from the application code using a function named `phy_get_romfuncs`.

```
// Disable both Bluetooth and Radio Module
esp_bt_controller_shutdown();
// Flush calibration data
esp_phy_erase_cal_data_in_nvs();
// Enabling Radio Module to force a new calibration
esp_phy_enable();
```
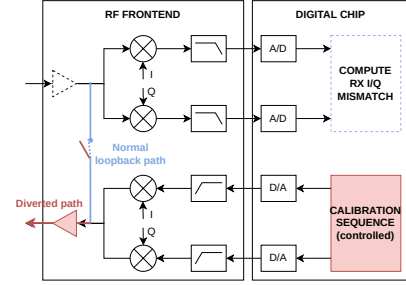Listing 6: Code snippet allowing to trigger a full calibration.

Fig. 8. Diverted calibration process.

As a result, we can re-use the hooking strategy presented in subsection V-A to intercept the function calls and execute arbitrary code.

Typical digital calibration techniques use a loopback between TX and RX path to estimate and compensate I/Q mismatch [39]. According to the low-level functions we identified, a similar strategy is implemented in ESP32 SoCs. Especially, we found a function named `rom_loopback_mode_en`, allowing us to enable or disable the loopback mode. We can then hook this function and run an infinite loop, preventing the loopback mode to be enabled and forcing the calibration sequence (e.g., a sine wave) to be transmitted over the air.

a) Controlling the signal: We can then directly manipulate the signal transmitted over the air from the software, acquiring fine-grained control over the transceiver. We can control the signal frequency by disabling the hardware control of the oscillator (using `phy_dis_hw_set_freq`) and providing a frequency offset in MHz from 2400MHz using `set_chan_freq_sw_start`. The switch time between two frequencies has been empirically evaluated using a HackRF One (with a sample rate of 10 Msps) to 83μs. Similarly, manipulating the parameters of `ram_start_tx_tone` allows generating a wide variety of signals, from a basic sine wave to invasive glitches (see appendix D).

b) Offensive use: This fine-grained control over the transmitted signal allows the implementation of multiple attacks, such as covert channels or jamming. The very fast switching time, in particular, allows to implement powerful jamming primitives. As a proof of concept, we implemented a BLE jammer, transmitting a glitched signal and hopping rapidly all along the three advertising channels, respectively, at 2402, 2426, and 2480MHz. This implementation is presented in Listing 7. We managed to prevent the reception of any advertisement packets by the surrounding devices. Since these channels are needed both for advertising and for initiating connections, jamming them simultaneously impacts most of the protocol functionality.

```
while (jammer_enabled) {
    // Configures the frequency to 2402 MHz (ch. 37)
    set_chan_freq_sw_start(2,0,0);
    // Generate a glitched signal
    ram_start_tx_tone(1,0,10,0,0,0);
    // Configures the frequency to 2426 MHz (ch. 38)
    set_chan_freq_sw_start(26,0,0);
    // Generate a glitched signal
    ram_start_tx_tone(1,0,10,0,0,0);
    // Configures the frequency to 2480 MHz (ch. 39)
    set_chan_freq_sw_start(80,0,0);
    // Generate a glitched signal
    ram_start_tx_tone(1,0,10,0,0,0);
}
```

Listing 7: Code snippet allowing to jam the three advertising channels simultaneously.

VI. DISCUSSIONS

In this article, we provided a large overview of the low-level internals of a popular embedded BLE stack. We subverted several software and hardware components to implement advanced offensive techniques. We were able to implement multiple offensive primitives, allowing complex wireless attacks targeting the BLE protocol but also various wireless protocols sharing physical layers similarities. The feasibility of implementing such attacks on a cheap and popular SoC underlines the urgency of analyzing and improving the security of wireless communication protocols co-existing in the 2.4GHz ISM band.

Indeed, the increasing use of connected devices in our daily life resulted in a chaotic deployment of heterogeneous wireless technologies, co-existing in the same environments and using similar physical layers. This co-existence introduces new security threats, such as cross-protocol attacks. In particular, the massive deployment of Bluetooth Low Energy in connected objects, smartphones, and laptops, makes it an ideal vector for targeting sensitive communications relying on insecure wireless protocols. Many proprietary protocols relying on weak designs deployed in the wild induce a significant risk for users. It exposes various commercial products to these attacks, including wireless keyboards and health-oriented sensors. We identified serious design weaknesses in the design of ANT and Riitek protocols and were able to exploit them to conduct eavesdropping and injection cross-protocol attacks targeting commercial products.

Two main offensive scenarios could leverage these low-level offensive capabilities. First, it is possible to implement a cheap and mobile cross-protocol attack platform based on an ESP32 SoC, similar to the Goodwatch project initiated by Travis Goodspeed [19]. We started the implementation of such a device during this research by writing a custom firmware for an ESP32-based smartwatch intended for IoT developers, the Lilygo T-Watch. Second, a connected device embedding an ESP32 SoC could be compromised using a code execution vulnerability or a malicious firmware update. It may allow an attacker to conduct malicious operations, such as a logical bomb triggering a massive jamming campaign or spying on sensitive communications. While the deployment of ESP32 SoCs within commercial devices seems relatively limited today (although it does exist, including for industrial solutions [25], [41]), it is particularly popular in the circle of hobbyists. It provides many attractive features for IoT developers and manufacturers.

VII. CONCLUSION

In this paper, we explored in depth the internals of the BLE stack embedded in ESP32, ESP32-S3, and ESP32-C3 SoCs, from its software architecture to its hardware components. We showed the feasibility of diverting their low-level functionalities to implement a wide set of advanced wireless attacks. We managed to manipulate the traffic processed by the Link Layer of the BLE protocol, but also to impact other wireless communication protocols which are not natively supported by the SoCs, but share similarities in terms of the physical layer and coexist in the same frequency band.

We also highlighted serious security risks related to the chaotic deployment of proprietary wireless communication protocols in the 2.4 GHz ISM band. In particular, we discovered or confirmed critical security flaws in the design of several of them, such as ANT, MosArt, and Riitek. We were able to conduct high-impact attacks targeting real-life devices manipulating sensitive data, such as wireless keyboards or heart rate monitors. The attack surface exposed by these insecure protocols is especially problematic, considering that they may be attacked from Bluetooth Low Energy devices, which are massively deployed in our daily lives. The proximity of the physical layers used by these different protocols and their coexistence within the same environments makes cross-protocol attacks possible.

In future work, we plan to explore two complementary axes. The low-level manipulation of Link Layer PDUs developed during this research opens new offensive perspectives targeting the BLE protocol. Implementing a fingerprinting approach for BLE devices, in particular, allows us to consider the development of an automatic vulnerability detection and exploitation approach within these devices' application and protocol stacks. The second axis will focus on the generalization of cross-protocol attacks leveraging the proximity of the physical layers within heterogeneous wireless protocols, to better understand and anticipate this new kind of threat.

REFERENCES

[1] "Bluedroid presentation page," https://source.android.com/docs/core/connect/bluetooth, Android Open Source Project.

[2] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, "Bias: Bluetooth impersonation attacks," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2020.

[3] D. Antonioli, N. O. Tippenhauer, and K. B. Rasmussen, "The knob is broken: Exploiting low entropy in the encryption key negotiation of bluetooth br/edr," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1047–1061.

[4] "Nimble github repository," https://github.com/apache/mynewt-nimble, Apache MyNewt.

[5] *Bluetooth Core Specification*, Bluetooth SIG, 07 2021, rev. 5.3.

[6] S. Bratus, T. Goodspeed, A. Albertini, and D. S. Solanky, "Fillory of PHY: Toward a periodic table of signal corruption exploits and polyglots in digital radio," in *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. Austin, TX: USENIX Association, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/woot16/workshop-program/presentation/bratus

[7] D. Cauquil, "Radiobit, a BBC Micro:Bit RF firmware," 2017, https://github.com/virtualabs/radiobit.

[8] ——, "Weaponizing the bbc micro bit," https://media.defcon.org/DEF%20CON%2025/DEF%20CON%2025%20presentations/DEF%20CON%2025%20-%20Damien-Cauquil-Weaponizing-the-BBC-MicroBit-UPDATED.pdf, 2017.

[9] ——, "You'd better secure your BLE devices or we'll kick your butts !" in *DEF CON*, vol. 26, 2018, available at https://media.defcon.org/DEFCON26/DEFCON26presentations/DEFCON-26-Damien-Cauquil-Secure-Your-BLE-Devices-Updated.pdf.

[10] R. Cayre, D. Cauquil, and A. Francillon, ""ESPwn32: Hacking with ESP32 System-on-Chips" artifacts for 17th IEEE Workshop on Offensive Technologies (WOOT'23)." Mar. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7786224

[11] R. Cayre, F. Galtier, G. Auriol, V. Nicomette, M. Kaâniche, and G. Marconato, "InjectaBLE: Injecting malicious traffic into established Bluetooth Low Energy connections," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2021)*, Taipei (virtual), Taiwan, Jun. 2021. [Online]. Available: https://hal.laas.fr/hal-03193297

[12] ——, "WazaBee: attacking Zigbee networks by diverting Bluetooth Low Energy chips," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2021)*, Taipei (virtual), Taiwan, Jun. 2021. [Online]. Available: https://hal.laas.fr/hal-03193299

[13] R. Dawes, "LogiTacker GitHub Repository," 2019, available at https://github.com/RoganDawes/LOGITacker.

[14] Ebiroll, "Github repository of xtensa ghidra extension," https://github.com/Ebiroll/ghidra-xtensa.

[15] N. F. for eeNews Europe, "Espressif moves exclusively to risc-v," https://www.eenewseurope.com/en/espressif-moves-exclusively-to-risc-v/, 2022.

[16] M. E. Garbelini, V. Bedi, S. Chattopadhyay, S. Sun, and E. Kurniawan, "BrakTooth: Causing havoc on bluetooth link manager via directed fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1025–1042. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/garbelini

[17] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sumei, and E. Kurniawan, "Sweyntooth: Unleashing mayhem over bluetooth low energy," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 911–925. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/garbelini

[18] T. Goodspeed, "Apimote ieee 802.15.4/zigbee sniffing hardware," https://www.riverloopsecurity.com/projects/apimote/.

[19] ——, "Goodwatch project website," https://kk4vcz.com/goodwatch/.

[20] ——, "Promiscuity is the nrf24l01+'s duty," http://travisgoodspeed.blogspot.com/2011/02/promiscuity-is-nrf24l01s-duty.html.

[21] T. Goodspeed, S. Bratus, R. Melgares, R. Shapiro, and R. Speers, "Packets in packets: Orson welles' In-Band signaling attacks for modern radios," in *5th USENIX Workshop on Offensive Technologies (WOOT 11)*. San Francisco, CA: USENIX Association, Aug. 2011. [Online]. Available: https://www.usenix.org/conference/woot11/packets-packets-orson-welles-band-signaling-attacks-modern-radios

[22] IEEE, "Ieee standard for low-rate wireless networks," *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pp. 1–709, April 2016.

[23] D. I. Inc., "Ant message protocol and usage, rev 5.1," https://www.thisisant.com/.

[24] S. Lab, "Bluetooth experimentation framework for broadcom and cypress chips." https://github.com/seemoo-lab/internalblue, 2020.

[25] Moduino, "Moduino x series - industrial iot controller based on esp32," https://moduino.techbase.eu.

[26] M. Newlin, "MouseJack : White Paper," in *DEF CON*, vol. 24, 2016, available at https://github.com/BastilleResearch/mousejack/blob/master/doc/pdf/DEFCON-24-Marc-Newlin-MouseJack-Injecting-Keystrokes-Into-Wireless-Mice.whitepaper.pdf.

[27] ——, "RFStorm nRF24LU1+ Research Firmware GitHub repository," 2016, https://github.com/BastilleResearch/nrf-research-firmware.

[28] Renesas, "Da14681 datasheet," https://www.renesas.com/us/en/document/dst/da14681-datasheet.

[29] T. Schroeder and M. Moser, "Keykeriki resources," 2010, available at http://www.remote-exploit.org/articles/keykeriki_v2_0__8211_2_4ghz/.

[30] M. Schulz, D. Wegemer, and M. Hollick, "The nexmon firmware analysis and modification framework: Empowering researchers to enhance wi-fi devices," *Computer Communications*, vol. 129, pp. 269–285, 2018.

[31] D. Spill, "Ubertooth One website," 2012, http://ubertooth.sourceforge.net/.

[32] S. Surminski, C. Niesler, F. Brasser, L. Davi, and A.-R. Sadeghi, "Realswatt: Remote software-based attestation for embedded devices under realtime constraints," in *CCS '21: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM, November 2021, pp. 2890–2905, event Title: 2021 ACM SIGSAC Conference on Computer and Communications Security. [Online]. Available: http://tubiblio.ulb.tu-darmstadt.de/129997/

[33] E. Systems, "Esp32-c3 series datasheet, version 1.4," https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf.

[34] ——, "Esp32-s3 series datasheet, version 1.6," https://www.espressif.com/sites/default/files/documentation/esp32-s3_datasheet_en.pdf.

[35] ——, "Esp32 series datasheet, version 4.2," https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[36] ——, "Espressif iot development framework. official development framework for espressif socs," https://github.com/espressif/esp-idf.

[37] ——, "Espressif soc serial bootloader utility," https://github.com/espressif/esptool.

[38] M. Vanhoef and F. Piessens, "Advanced wi-fi attacks using commodity hardware," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 256–265. [Online]. Available: https://doi.org/10.1145/2664243.2664260

[39] I. Vassiliou, K. Vavelidis, T. Georgantas, S. Plevridis, N. Haralabidis, G. Kamoulakos, C. Kapnistis, S. Kavadias, Y. Kokolakis, P. Merakos, J. Rudell, A. Yamanaka, S. Bouras, and I. Bouras, "A single-chip digitally calibrated 5.15-5.825-ghz 0.18-um cmos transceiver for 802.11a wireless lan," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 12, pp. 2221–2231, 2003.

[40] J. Wright, "Killerbee: Practical zigbee exploitation framework," https://www.willhackforsushi.com/presentations/toorcon11-wright.pdf, 2009.

[41] Zerynth, "Industrial iot device - 4zerobox," https://zerynth.com/products/hardware/4zerobox.

[42] *ZigBee Specification*, https://zigbeealliance.org/wp-content/uploads/2019/11/docs-05-3474-21-0csg-zigbee-specification.pdf, Zigbee Alliance, 2015.

APPENDIX A
ANT NETWORK KEY ALGORITHMS SPECIFICATION

The algorithm is split into two main steps:

- the key validation, allowing to check if the key is valid
- the preamble derivation, allowing to generate a two-byte long preamble from the key

The key is considered valid if a value derived from the key equals zero. This value comprises a series of expressions linked to the key, which are then combined using a bit-wise *or* operator.

- The first expression takes the 3rd byte of the key, applies a mask of value 0xec using a bit-wise *and* operator, then is xored with value 0x20
- The second expression takes the 3rd and 4th bytes of the key, xor them together, applies a mask of value 0x3f using a and bitwise operator, then is xored with value 0x1a
- The third expression takes the 3rd, the 4th, and the 5th bytes of the key, xor them together, applies a mask of value 0xd7 using a bit-wise *and* operator, then is xored with value 0x47
- The fourth expression takes the 3rd, the 4th, and the 5th and the 6th bytes of the key, xor them together, applies a mask of value 0xdb using a bit-wise *and* operator, then is xored with value 0x11
- The fifth expression takes the 3rd, the 4th, the 5th, the 6th, and the 7th bytes of the key, xor them together, applies a mask with value 0x79 using a bit-wise *and* operator, then is xored with value 0x50
- The sixth expression takes the 3rd, the 4th, the 5th, the 6th, the 7th, and the 8th bytes of the key, xor them together, applies a mask of value 0xf7 using a and bit-wise operator, then is xored with value 0x93
- The seventh expression takes the 2nd, 3rd, 4th, 5th, 6th, 7th, and 8th bytes of the key, xor them together, applies a mask of value 0xbe using a bit-wise and operator, then is xored with value 0x36
- The last expression takes every byte of the key, xor them together, applies a mask of value 0xef using a bit-wise *and* operator, then is xored with value 0x8f

If the key is valid, the preamble is then generated by generating two bytes derived from the key:

The least significant byte is formed by combining several expressions using a bit-wise *or* operation:

- the first expression takes the 2nd, 3rd, 4th, 5th, 6th, 7th, and 8th bytes, xor them together, and applies a mask with value 0x41 using a bit-wise *and* operator
- the second expression takes every byte of the key, xor them together and applies a mask of value 0x10 using a bit-wise *and* operator
- the third expression takes the 3rd, 4th, and 5th bytes of the key, xor them together and applies a mask of value 0x28 using a bit-wise *and* operator
- the fourth expression takes the 3rd, 4th, 5th, 6th, and 7th bytes of the key, xor them together and applies a mask with value 0x86 using a bit-wise *and* operator

the most significant byte is formed by combining several expressions using a bitwise *or* operation:

- the first expression takes the 3rd, the 4th, the 5th, the 6th, the 7th, and the 8th bytes, xor them together, and applies a mask of value 0x08 using an and bitwise operator
- the second expression takes the 3rd and the 4th bytes of the key, xor them together, and applies a mask of value 0xc0 using an and bit-wise operator
- the third expression takes the 3rd byte of the key and applies a mask of value 0x13 using an and bit-wise operator
- the fourth expression takes the 3rd, the 4th, the 5th, and the 6th bytes of the key, xor them together and applies a mask of value 0x24 using a bit-wise *and* operator

APPENDIX B
ANT NETWORK KEY ALGORITHMS INDEPENDENT IMPLEMENTATION

```c
#include <stdio.h>
#include <stdint.h>
#include <assert.h>
#include <stdbool.h>

// We assume the CPU is running in little-endian mode.

// Useful structures
struct key {
    union {
        uint8_t bytes[8];
        uint64_t raw;
    };
};
```

```c
struct key_preamble {
    union {
        struct {
            uint8_t low;
            uint8_t high;
        };
        uint16_t raw;
    };
};


// Magic value tables

// Validation function
static uint8_t validate_xor_table[8] = {0x20, 0x1a, 0x47, 0x11, 0x50, 0x93, 0x36, 0x8f};
static uint8_t validate_and_table[8] = {0xec, 0x3f, 0xd7, 0xdb, 0x79, 0xf7, 0xbe, 0xef};

// Preamble gen function
static uint8_t preamble_xor_key_table[8] = {0xfe, 0xff, 0x1c, 0x7c, 0xfc, 0x0c, 0x04, 0x3c};
static uint8_t preamble_and_table[8] = {0x41, 0x10, 0x28, 0x86, 0x08, 0xc0, 0x13, 0x24};

// Validate the key
static bool is_valid_key(struct key k) {
    // Xor index variables
    const uint8_t xor_start_offset = 2;
    uint8_t nb_xor = 1;

    // Key as bytes
    uint8_t* k_b = k.bytes;

    uint8_t k_deriv = 0;
    uint8_t tmp;

    for (uint8_t i = 0; i < 8; ++i) {
        tmp = 0;
        // Xor key bytes
        for (uint8_t j = 0; j < nb_xor; ++j) {
            if (j < 6) {
                tmp ^= k_b[xor_start_offset + j];
            } else if (j == 6) {
                tmp ^= k_b[1];
            } else {
                tmp ^= k_b[0];
            }
        }

        // And op
        tmp &= validate_and_table[i];

        // Xor op
        tmp ^= validate_xor_table[i];

        k_deriv |= tmp;
        ++nb_xor;
    }

    return k_deriv == 0;
}

// Generate the key's preamble
// Precondition: the key should be valid
static uint16_t gen_key_preamble(struct key k) {
    struct key_preamble k_p = {0};

    // Key as bytes
    uint8_t* k_b = k.bytes;

    uint8_t tmp = 0;
    uint8_t tmp2;

    for (uint8_t i = 0; i < 8; ++i) {
        tmp2 = 0;
        if (i == 4) {
            k_p.low = tmp;
            tmp = 0;
        }

        for (uint8_t j = 0; j < 8; ++j) {
            if (preamble_xor_key_table[i] & (1 << j)) {
                tmp2 ^= k_b[j];
            }
        }
```

```
        tmp2 &= preamble_and_table[i];
        tmp |= tmp2;
    }

    k_p.high = tmp;

    return k_p.raw;
}

int main() {
    struct key k;

    printf("Key: ");
    assert(scanf("%lx", &k.raw) == 1);
    printf("\n");

    if (is_valid_key(k)) {
        uint16_t k_preamble = gen_key_preamble(k);

        printf("The provided key is correct, and the corresponding preamble is: 0x%x\n", k_preamble);
    } else {
        printf("Error: invalid key\n");
        return 1;
    }

    return 0;
}
```

APPENDIX C
Rii i8 Wireless Mini Keyboard & Touchpad protocol overview

A. Packet and payload formats

TABLE I
PACKET FORMAT

Preamble	Address	Unknown	Seq. number	Payload	CRC
0x000000AA	5 bytes	4 bits	4 bits	variable	2 bytes

TABLE II
KEYBOARD PAYLOAD FORMAT

Frame type	HID Data
0x0B	7 bytes

TABLE III
MOUSE PAYLOAD FORMAT

Frame type	Button action	X	Y	Padding
0x04	0x01 : left, 0x02 : right	2 bytes (LE)	2 bytes (LE)	0x0000

B. CRC computation algorithm

CRC field is 2 bytes long and is calculated over the header and payload fields, using a polynomial equal to 0x1021 and an initialization value equal to 0x8b83.

APPENDIX D
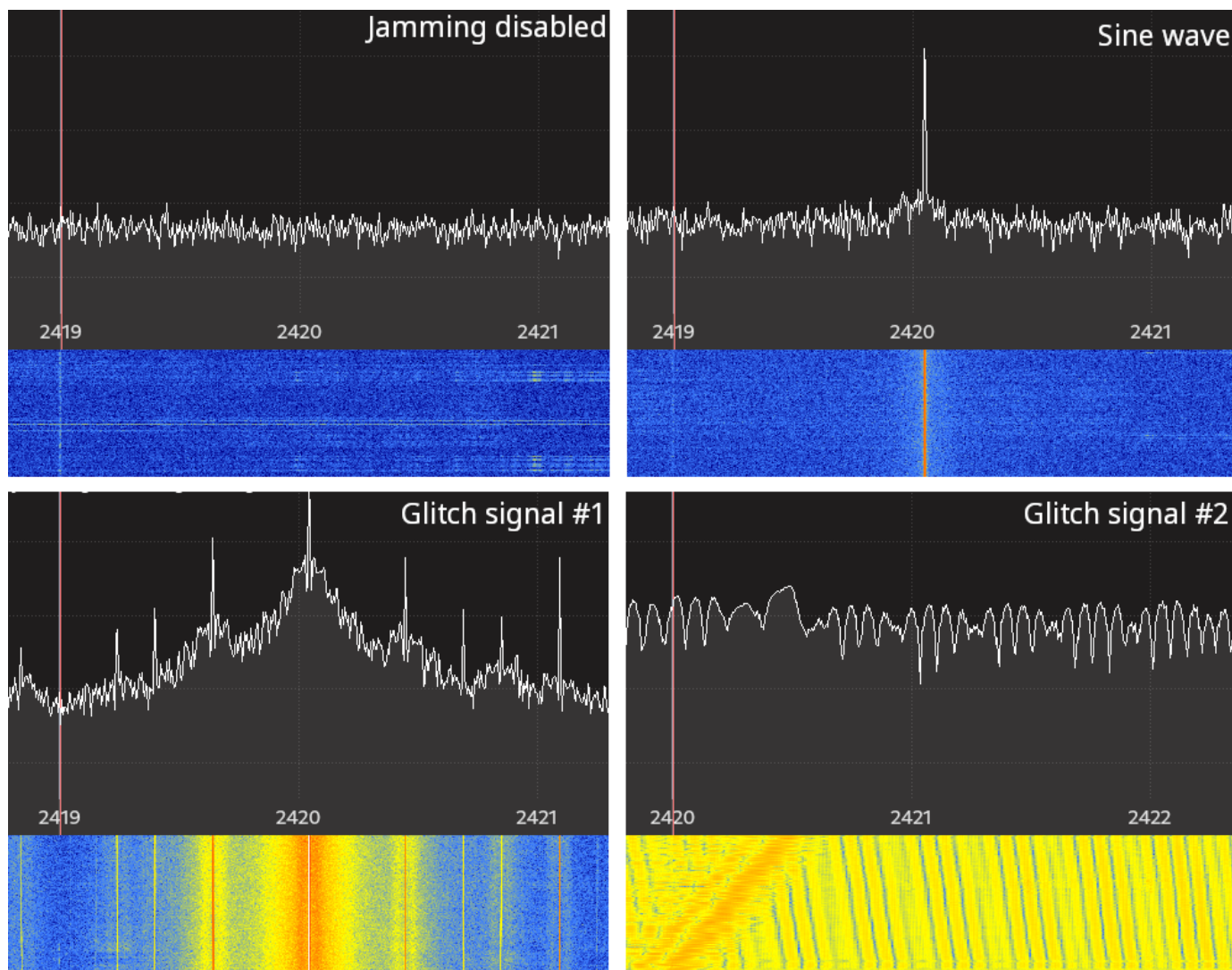## EXAMPLES OF SIGNALS GENERATED BY DIVERTING THE CALIBRATION PROCESS.

Fig. 9. Samples of malicious signals generated from ESP32, collected with HackRF One and GQRX.