École Polytechnique Fédérale de Lausanne

Support Mynewt Nimble on HALucinator

by Duo Xu

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Dr. Gwangmu Lee
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 10, 2022

Abstract

Bluetooth Low Energy (BLE) is now vastly used. But very not many works have been proposed to offer full protocol fuzzing testing for BLE stack. Firmware rehosting, as a powerful tool for firmware emulation and dynamic analysis, can be of great help in fuzzing testing BLE stack. In this project, we add support of Mynewt RTOS and Nimble on HALucinator, and validate them through one blinky and one iBeacon project, which lays a foundation for future work on fuzzing Nimble and other BLE libraries. As a result, HALucinator now have good support for Mynewt RTOS, and for Nimble's advertising functionality.

Contents

Chapter 1

Introduction

Designing energy-efficient communication protocol for low-power devices (i.e, devices with limited power supply and limited memory space) is vital to defer battery replacement for such devices as much as possible. To that end, *Bluetooth Low Energy* (BLE) is a key wireless communication protocol for such devices.

This class of devices has universal use-cases. Namely, devices such as electronic locks, alarm systems, process monitoring or medical devices where having a high level of safety and security is the main requirement. BLE provides the possibility of controlling and monitoring these devices by smartphones or laptops.

The wireless characteristic of BLE makes the life of an attacker "Eve" easier, because she does not need to have physical access to a device with BLE interface for starting an attack, and the risk of being detected during the attack is low. As a result, securing BLE interfaces is a major challenge. Potential attack targets include sniffing, denial of service (DoS), spoofing, injection of information, partial or complete takeover of connections, tracking, and location. The Bluetooth specification leverages techniques such as multiple device pairing schemes, optional encryption and authentication of the connection, or address randomization to countermeasure these threats. However, these techniques are effective only if properly specified, used, and implemented. There were flaws and serious weaknesses in earlier versions of the definition of these security measures that any implementer should be aware of.

Currently, however, not so many work has been done on BLE full stack fuzzing testing. In previous work on Bluetooth fuzz testing, some work [1, 6] only supported L2CAP and ATT layers, and the validity of the fuzz tests used was not guaranteed by reasonable test optimization. In InternalBlue [7], the low-level implementation of Bluetooth is investigated and allows sniffing and injection of BLE packets. But InternalBlue can only work when a peripheral device is connected. In addition, the number of fields that can be accessed in the packet is limited. In [5], they propose the first comprehensive approach for BLE testing, which is not limited to specific layers. This system design offers more flexibility for fuzzing testing arbitrary BLE protocol implementations.

One problem with the system brought up in [5] is that their method bases heavily on using real hardware and the state machine is not flexible enough to test the full stack of a BLE library. Those two deficiencies can result in some problems for testing a specific BLE stack implementation. For example, it may not be so easy to get a board that supports the targeting BLE library. Also, since they can only interact according to the state machine, they will lack the freedom to test certain functions. Besides, it is not easy to include all the APIs in one firmware, which will make the testing less comprehensive, so some changes of the program under test are needed. But due to the usage of real hardware, each time the new firmware has to be rebuilt and flushed onto the board, which will reduce the fuzzing efficiency.

To address the aforementioned problems, we propose to use rehosting technique to offer full stack emulation and fuzzing. Rehosting is a process to move software stacks from physical systems into virtual environments to offer sufficient modeling of hardware behavior. It has made great impacts in the world of security analysis. With the emulation of real hardware, researchers have more freedom to carry out dynamic analysis for their firmware, without the restriction resulted from not having proper board support. So far, many rehosting systems have been proposed. For example, there are Avatar^2 [9], HALucinator [2], PANDA [3], P^{2}IM [4], etc. Detailed descriptions about these systems are in the next section. In this project we will use HALucinator as the base emulator for the BLE firmware. It addresses the challenge of peripheral accessing by replacing the Hardware Abstraction Layer (HAL) functions.

In this project, the targeting BLE stack library is Mynewt Nimble. One important reason is because it is open-source. With all the open-source functions, it would be better for us to carry out the rehosting for related HAL functions. But this may also induce some problems to solve. Firstly, due to the Real-Time Operating System nature of Mynewt's OS, timing becomes more crucial. All the task switching and events requiring real-timing would need to have more precise timing to make them work. Besides, with the context switching feature, the Systick interrupt and the interrupts needed for context change may occur together and the nested interrupts should be well-dealt with. Besides, rehosting a firmware based on RTOS is more complicated than simply offering the functionality. For example, when rehosting a simple blinky project that uses GPIO to set the LEDs, for a simple firmware we can just rehost the GPIO calling functions and that would be sufficient. For a blinky project with RTOS, however, we should also make sure all the tasks and events are well set up, which may include the examination of lots of other OS library functions. Moreover, since the ultimate goal is to test the Nimble library, simply make the firmware run without crash and accord to the program specification is not enough. We should make sure the functions we've rehost are only the necessary ones.

As a result, I successfully rehosted one Mynewt blinky project, and one iBeacon Nimble project. The emulation of the two projects is well validated, which means that we now have good support for Mynewt RTOS and Nimble projects on HALucinator.

Chapter 2

Background

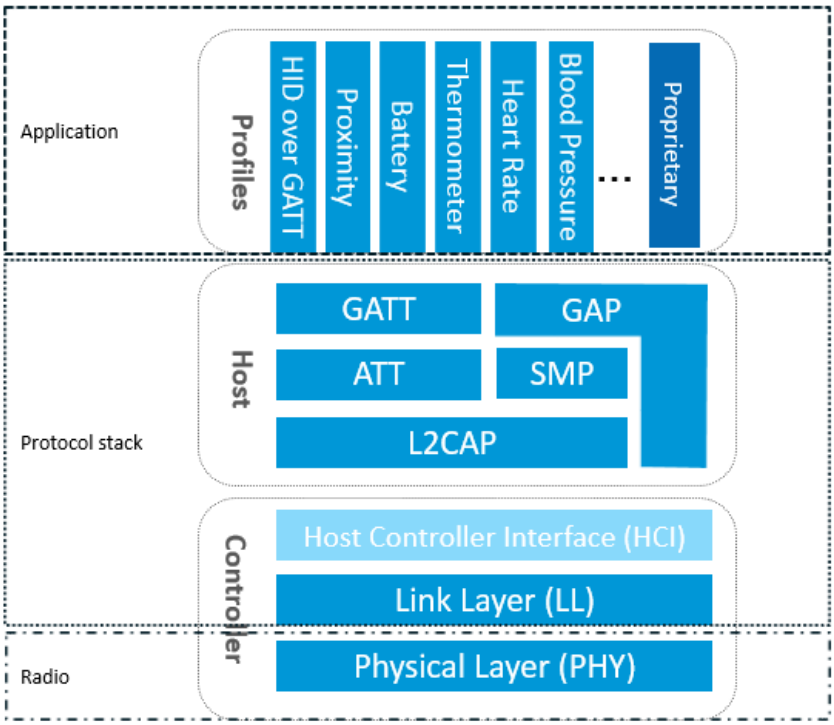

Figure 2.1: BLE protocol stack

2.1 BLE

BLE is mainly a good fit in a network of some low-power embedded systems (e.g., medical sensor) for obtaining a specific type of value and send it with a wireless communication to the user device.

In the following we discuss the main four steps of a typical communication procedure between a BLE device and a user device.

2.1.1 Advertising

Three radio frequency (RF) channels of BLE device are dedicated to advertising. In this step, the BLE device periodically announces its presence by broadcasting “advertising” packets on these channels. Each packet includes a unique identifier of the BLE device and what kind of services it provides.

2.1.2 Connecting

The user device inspects the advertising channels to see if any BLE device is advertising itself. After finding a BLE device, if the user device wants to establish a connection, it sends a connection request packet. If the BLE device accepts the request, the two devices will start communicating over other channels for data transmission. After establishing the connection, the BLE device may or may not continue advertising.

2.1.3 Pairing

To establish a secure channel for communication, the BLE specification introduces multiple handshake methods. Through the pairing procedure, the BLE device and the user device can generate a shared cryptographic key to encrypt or authenticate later communicated payloads.

2.1.4 Accessing

In this step, the user device makes use of the service provided by the BLE device by getting data values from it.

2.2 Firmware rehosting

Plenty of rehosting systems have been brought up in recent years. Avatar^2 [9] is a dynamic multi-target orchestration and instrumentation framework which can be utilized for firmware analysis. Its main contribution is allowing for other tools (e.g. qemu, gdb, and etc.) to interact and transfer data. This method is based on hardware-in-the-loop emulation. HALucinator [2] has more flexibility offered by its ability to replace the Hardware Analysis Layer (HAL) functions. It

uses Avatar[2] and QEMU as its base emulators, which are used to hook and replace HAL functions. This tool bases mostly on the assumption that a majority of programmers use HAL functions for writing firmware programs. PANDA [3] also uses QEMU as the base emulator, it has a main advantage that it can record and reply executions. This can enable researchers to have a deeper and whole-system analysis.

2.3 RTOS

Operating System (OS) is the software that manages resources of the system. One of the jobs that an OS does, is to give the illusion of simultaneous execution of many processes at the same time. The part of OS that handles this job is the *scheduler*. The scheduler decides which process should run next on each core of the processor. General-purpose Operating System (GPOS) is the type of OS that is designed to operate for "general" users and to give them a nice user experience. Typically, in GPOS, the goal of the scheduler is to provide fairness between all the processes or staying responsive to the user. On the other hand, in Real-time Operating System (RTOS) the scheduler is designed to provide a deterministic and predictable scheduling. The goal for this type of scheduler is to meet *deadlines*. Namely, the system should respond to each event within a defined period of time. So, real-time means having a guarantee of meeting the deadlines by following a predictable behavior. Currently many RTOSes have been brought up, like Mynewt, FreeRTOS, RIOT, and etc.

Chapter 3

Emulation setup

This and the next sections describe how the problems are solved, which engineering problems need to be solved, and how they are solved. In [10], the authors divided the rehosting problem into three stages: "pre-emulation", "emulation", and "post-emulation". In this and the following section, we will follow this specification and present the difficulties encountered in the "pre-emulation" and "emulation" stages and how to solve them.

3.1 mynewt bootloader

Mynewt has a "bootloader" that loads the Mynewt OS image into memory and conducts some checks before allowing the OS to be run. It manages images for the embedded system and upgrades of those images using protocols over various interfaces (e.g. serial, BLE, etc.). Typically, systems with bootloaders have at least two program images coexisting on the same microcontroller, and hence must include branch code that performs a check to see if an attempt to update software is already underway and manage the progress of the process. After examining the source code of the bootloader, it seems that its functionality mainly targets at the image security checking and over-the-air updating. Therefore when rehosting Mynewt applications we only need to take the application image into consideration.

3.2 memory setup

The program's entry point and memory layout should be decided before loading. This can be done by reading the value from ELF file. Base address should be the starting address of the ".text" segment. The memory layout of ARM Cortex-M4 is shown in Figure 3.1. The memory range for peripheral is set accordingly, result of which is shown in the code below. In flash we should

offer the binary containing the code and data. Also set the permissions as "r-x" so that it is not writable. Some problem with avatar is that when memory range is set too large, the gdb protocol cannot be initialized. So we just use a more restricted size of ram here. Besides memory setup, we should also include the function addresses in the config files. This can be generated from

```
memories:
  flash: {base_addr: 0x0000c020, file: ble_app.elf.bin,
    permissions: r-x, size: 0x1fd00000}
  ram: {base_addr: 0x20000000, size: 0x51000}
peripherals:
  logger: {base_addr: 0x40000000, emulate: GenericPeripheral, permissions: rw-, size: 0x20000000}
```

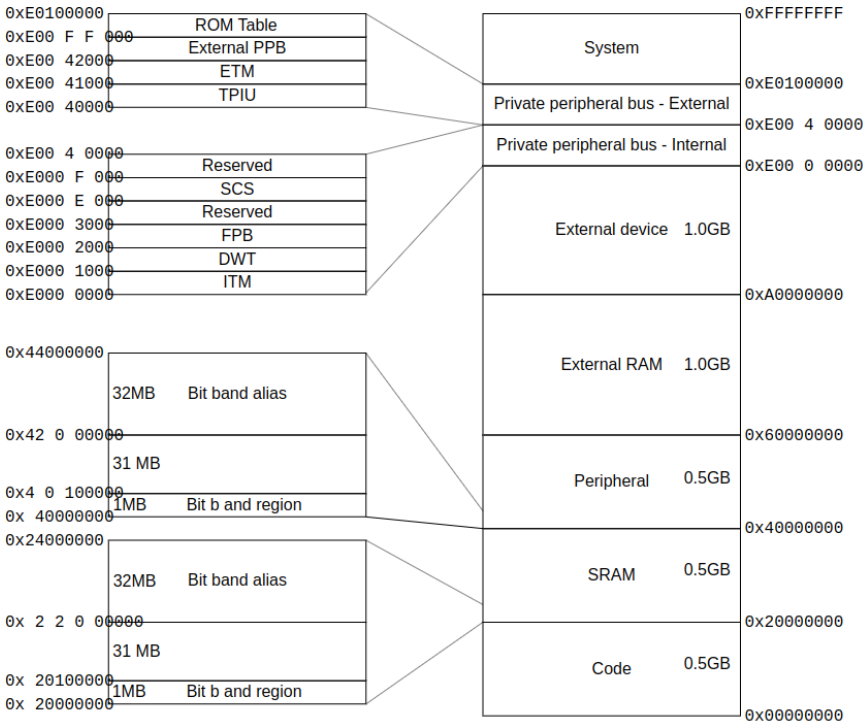

Figure 3.1: Arm Cortex-M4 memory layout

Chapter 4

Implementation

When using HALucinator for firmware rehosting, the most important part is designing breakpoint handlers. The whole designing procedure is debugging-oriented. Just fire HALucinator and wait till it crash or halt due to the lack of support for certain functions or missing important setups. This part is where human efforts are required the most. Making the firmware run without crashing is simple. We can just hook all the functions that crash the program and faking that it is well-handled. In this way, however, the rehosting would not be so useful since it is not executing as what we have expected. Therefore, digging a bit into the semantics of the functions is crucial to carry out meaningful rehosting, which can be used later for fuzzing or other activities. In this section I will elaborate on four important parts of Mynewt's programs for blinky and iBeacon projects, focusing on what is their original functionality and how we can handle them well.

4.1 Mynewt OS

First, I will introduce the basis of Mynewt OS. Understanding how this RTOS works is fundamental to make sure if our emulation carries out the OS setup properly. Like all the RTOS, Mynewt bases heavily on the concepts of task and event. Each task has an event queue for event organizing. The OS task scheduler maintains the list of tasks and decide which one should be run next. One of the core concepts in Mynewt OS is the callout, it is the OS timers that used to maintain part of the "real-time" feature, the callouts will require one event be added tasks have a timer indicating when they should be executed. The inaccuracy of timing can pose problems here. For example, if the timer device is not properly setup, some setup tasks may not be executed properly and in the end make the emulation have less fidelity.

Now that we figure out how Mynewt OS generally executes the tasks, it is the time to examine the functions needed for the operating system setup (i.e. initialization functions executed between the first function executed and main). The functions checked are shown in Table 4.1.

Function	Description	Rehosted
sbrkInit	setups memory	No
SystemInit	checks system initialization	No
hal_system_init	sets peripheral register structure	No
NVIC_Relocate	relocates NVIC table	No
start	starts the application	No
_libc_init_array	empty function	No
os_init	initializes the OS	No
os_start	starts the main function	No

Table 4.1: Functions in the OS setup process

Mostly the functions are just interacting with the memory, and even for the functions that have started the event queues and tasks do not have timing critical events. During the first few times of emulation, the crashes can happen during the initialization. But they are all caused by invalid memory accessing, by allocating more memory they are solved. Therefore, in the end none of the functions need to be rehosted.

4.2 Timer

The board used in this project is the nordic's nrf52840. The clock control system can source the system clocks from a range of internal or external high and low frequency oscillators and distribute them to modules based upon a module's individual requirements. Since we do not have the real timer device, we have to fake the signal sent out by the timer. The handler of timer signal is handled by the "Systick" interrupt handler. The important part is distinguishing where the timer is triggered. After examining the code, we can see that the timer is started by the "hal_system_clock_start" function, where they check if HFXO or LFXO is used and also make sure the system is in the correct state. Therefore the next step is using the peripheral model offered by HALucinator to emulate the timer device.

The usage of HALucinator's timer peripheral is straightforward. It mainly uses another class called "TimerIRQ", which is a subclass of Thread. By initializing its object and call the "start" function. It will start a new thread and periodically send out interrupt to Qemu with number 15. This will trigger the "Systick_Handler" and thus advance the OS's time. Before starting the timer, one other important aspect is to change the required frequency, it is unfeasible to follow the hardware timer's frequency. With the handler for the "hal_system_clock_start", we now consider all the initialization and setting for the timer is done. Therefore, when calling the rest clock initialization functions we only need to replace them with empty handler functions and claim it is well handled.

In the blinky example. Another function needs to be taken care of is the "os_time_delay". It will delay the current task for some time so that the task can be executed periodically. Same as

Function	Description
Systick	handler for the Systick interrupt
hal_system_clock_start	start the timer
os_time_delay	make the current task sleep for a certain period
os_cputime_get32	get lower 32 bits of the timer
nrf52_clock_hfxo_request	request for the HFXO clock

Table 4.2: Rehosted functions for timer

Function	Description
hal_gpio_init_out	initializes GPIO
hal_gpio_toggle	toggles GPIO port

Table 4.3: Rehosted functions for GPIO

before, in the emulation we cannot simply execute the function with the same delaying time. Instead we can just hook the function and change the value of the r0 register, which contains the input argument. This time the handler should allow the function to resume its execution. Summary of the functions is shown in Table 4.2

4.3 GPIO

For GPIO, the main functions need to be rehosted are just the two "hal_gpio_init_out" and "hal_gpio_toggle". Descriptions of them are shown in Table 4.3

4.4 BLE

In BLE, the functions need rehosted are mainly those used for host-controller syncing, the hci interaction functions, and the functions used for host to interact with the controller. Descriptions of the functions are shown in Table 4.4. Through rehosting the "ble_hs_startup_go" function, we can pretend that the host and controller syncing is already finished. In the real function it is achieved through some data communication between host and controller. In order to fake the

Function	Description
ble_hs_startup_go	syncs up BLE host and controller
ble_gatts_start	starts GATT service
ble_ll_rand_data_get	gets random data from hardware random number generator
ble_hs_hci_cmd_tx	transport cmd to controller
ble_ibeacon_set_adv_data	sets advertisement data

Table 4.4: Rehosted functions for BLE

advertising procedure, I implemented a "BLERadio" class, which is the subclass of the "Thread" class. Whenever the "ble_hs_hci_cmd_tx" function gets the command for enabling advertising, it will start running in a separate thread and printing out the advertisement data.

Chapter 5

Evaluation

Evaluation of the fidelity of rehosting is a challenge problem. It greatly depends on the intention of rehosting. In this section I will elaborate on how I evaluated my rehosting. Because of the complexity introduce by the RTOS, I mainly use manual efforts for execution path validation to ensure the targeting projects are well-rehosted. This is the most time-consuming part throughout the whole project, but it is the most reliable way I can think of for validating a project from a new platform. The validation part includes the validation of timer, RTOS , BLE.

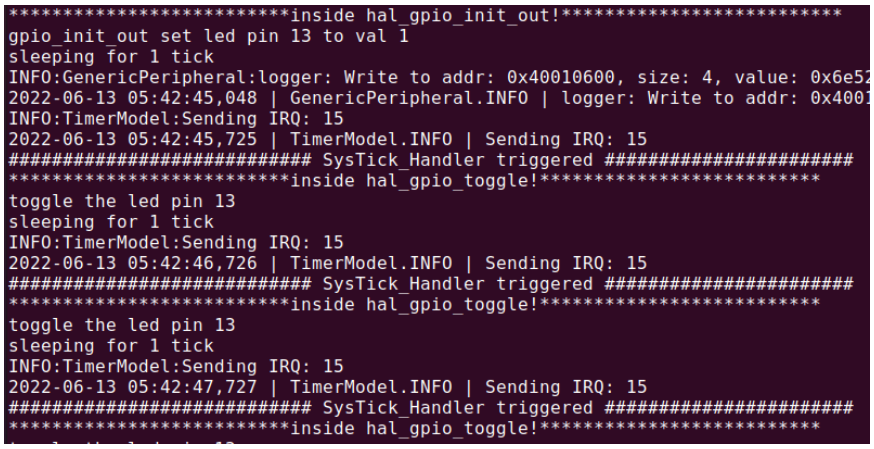

Figure 5.1: blinky rehosting validation

As for timer, we mainly need to care about if it has been successfully activated, and if the Systick interrupt has been signaled and handled. Besides, it is also important to check if the "os_time_delay" is functioning correctly (i.e. stall the task for exactly one tick). The rehosting result of the blinky project is shown in Figure 5.1. From the figure we can see that the "TimerModel" has been successfully initialized and functioning well, which means the "hal_system_start" function is handled correctly. Moreover, the "hal_gpio_toggle" is called at the same rate as the "Systick_Handler", which shows that the "os_time_delay" function is working properly. In the meanwhile, obviously the rehosting of GPIO APIs are correctly implemented.

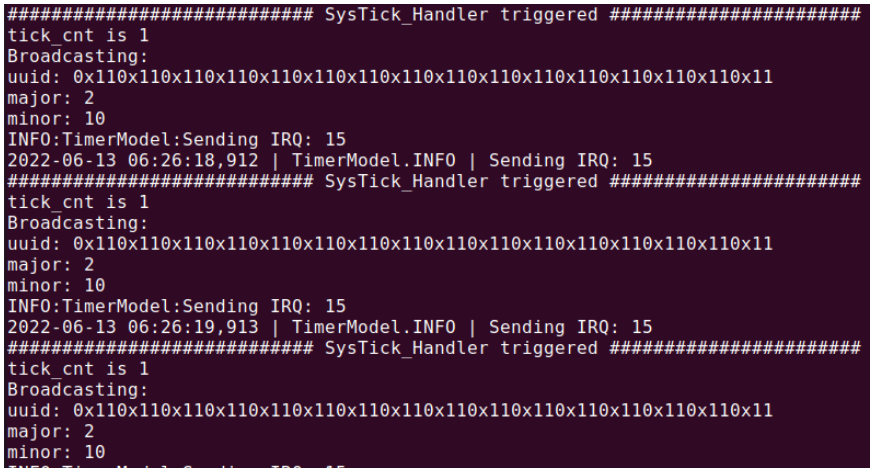

Figure 5.2: ibeacon rehosting validation

To validate that we have correctly rehosted the Mynewt RTOS, I will talk about how each core feature of Mynewt RTOS has been supported. Firstly, it is obvious that with the correct implementation of timer and Systick interrupt, the Callout feature will be well supported. The Callout is a way of setting up an OS time. When the timer fires, it is delivered as an event to task's event queue. In the "Systick_Handler", the "os_callout_tick" function is called to deal with all the callouts. Therefore, with the correct implementation of "Systick_Handler", the Callout feature is also taken care of. Secondly, in Mynewt, Scheduler's job is to maintain the list of tasks and decide which one should be running next. The fact that "hal_gpio_toggle" executes in the same frequency as Systick proves that the task can be correctly put to sleep and waken up, which shows the Scheduler feature of Mynewt is also supported. With the correct implementation of the Scheduler and the Callout features, the Task and Event Queues features are undoubtedly supported.

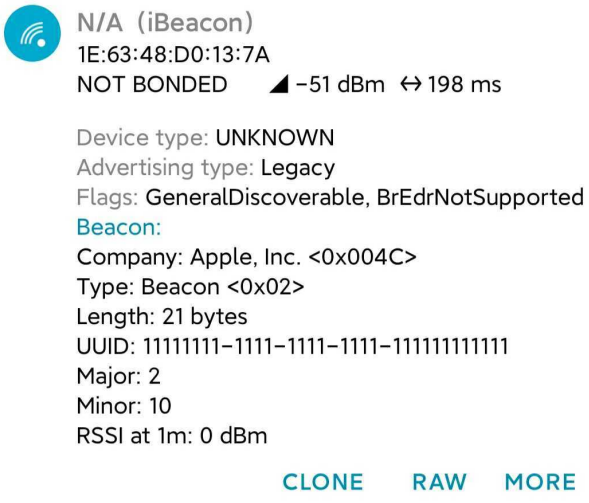

Figure 5.3: ibeacon advertisement

Validation of the rehosting for BLE is straightforward. The rehosted adverting ibeacon program and the real advertising messages are shown separately in Figure 5.2 and Figure 5.3. From the figures we can see that the advertisement messages are the same, which shows that the data have been set correctly for broadcasting. Besides, the periodic broadcasting log shows that the advertising command is well handled for the controller.

Chapter 6

Related Work

In BLE testing, the SweynTooth [5] tool offers good mechanism for carrying out fuzzing testing on BLE stack. It utilizes real hardware, and a state machine modeling the BLE protocol. During testing, it tries to either send out malformed packets or normal packets at the wrong time (or state). It achieves good results, but still has some deficiencies. For example, some hardware may not be easily accessible. Besides, state machine can be too coarse-grained for testing. Those problems can be solved through rehosting. Some other works [1, 8] have been proposed for BLE testing, but they either are not automated enough or have a restricted testing scope.

Chapter 7

Conclusion

In this project, we explored the Mynewt RTOS and Nimble libraries and utilized HALucinator for rehosting the blinky and iBeacon projects. In the end we validate that now HALucinator has good support for Mynewt RTOS and the advertisement of Nimble library. This can lay a foundation for future work on BLE fuzzing testing. But some problems are still to be solved. For example, to fully support RTOS even in racing scenarios, we should have good implementation to deal with the racing conditions and the inaccuracy of virtual timing. Besides, nested interrupts should also be supported to have more fidelity in our emulation.

Bibliography

[1] Hou-Fu Cheng and Yu-Qing Zhang. “Bluetooth OBEX vulnerability discovery technique based on fuzzing”. In: *Computer Engineering* 34.19 (2008), pp. 151–153.

[2] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. “{HALucinator}: Firmware Re-hosting Through Abstraction Layer Emulation”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1201–1218.

[3] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. “Repeatable reverse engineering with PANDA”. In: *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. 2015, pp. 1–11.

[4] Bo Feng, Alejandro Mera, and Long Lu. “{P2IM}: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1237–1254.

[5] Matheus E Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. “{SweynTooth}: Unleashing Mayhem over Bluetooth Low Energy”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 2020, pp. 911–925.

[6] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. “Poster: Iotcube: an automated analysis platform for finding security vulnerabilities”. In: *Proceedings of the 38th IEEE Symposium on Poster presented at Security and Privacy*. 2017.

[7] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. “InternalBlue-Bluetooth binary patching and experimentation framework”. In: *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. 2019, pp. 79–90.

[8] Gianluigi Me. “Exploiting buffer overflows over Bluetooth: the BluePass tool”. In: *Second IFIP International Conference on Wireless and Optical Communications Networks, 2005. WOCN 2005*. IEEE. 2005, pp. 66–70.

[9] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. “Avatar 2: A multi-target orchestration platform”. In: *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)* Vol. 18. 2018, pp. 1–11.

[10] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. “Challenges in firmware re-hosting, emulation, and analysis”. In: *ACM Computing Surveys (CSUR)* 54.1 (2021), pp. 1–36.