VaktBLE: A Benevolent Man-in-the-Middle Bridge to Guard against Malevolent BLE Connections

1. Abstract

This artifact showcases the implementation of *VaktBLE* paper and contains instructions to deploy and evaluate a real-time software and hardware platform to defend Bluetooth Low Energy (BLE) peripherals against over-the-air attacks. Such attacks may exploit known and potentially unknown vulnerabilities. The platform is designed for two setups: an anchored setup using a PC (x86_64) and a portable setup using an embedded Linux system (Orange Pi Zero 3 with ARM Cortex-A53). Since the framework interacts with Bluetooth Low Energy devices remotely, access to a remote machine is provided through AnyDesk. Subsequently, we describe the detailed experimental procedures to aid in reproducing the evaluation results of *VaktBLE* paper, in addition to including the experimental data obtained during our evaluation. To this end, the artifact also includes scripts to launch attacks against evaluated BLE devices attached to the remote machine.

2. Hardware and Software Requirements for *VaktBLE*

The artifact showcases the capabilities of our framework and how it (almost) deterministically hijacks BLE connections by abusing the start of it to protect Commercial-off-the-shelf (COTS) devices. To facilitate testing, the evaluation machine already has all the system libraries and python dependencies required to run *VaktBLE*. However, instructions to build from source are included in Section 2.5.

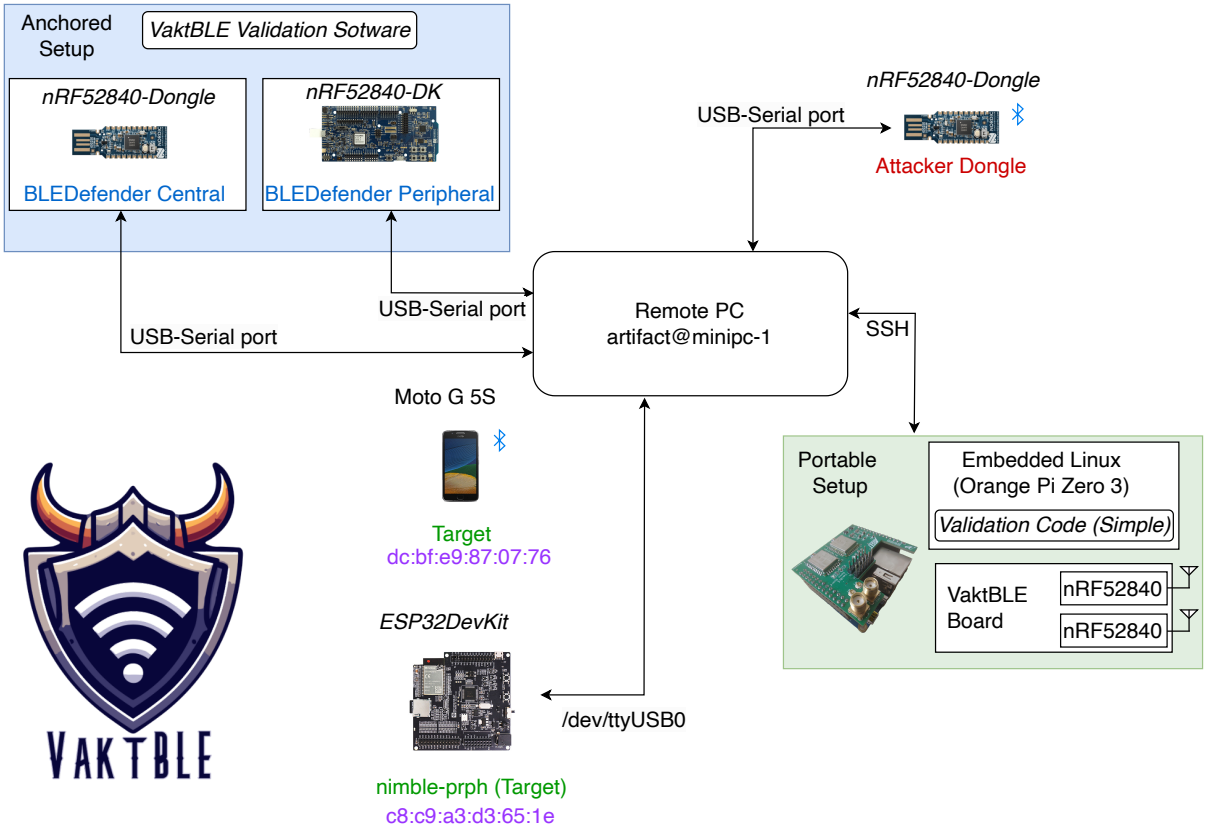


Figure 1: Overview of *VaktBLE* Evaluation Setup.

2.1. Hardware Dependencies

- nRF52840-DK - MitM Peripheral (non-compliant)
- nRF52840-Dongle - MitM central (non-compliant)
- ESP32 DevKit - Vulnerable BLE target
- nRF52840-Dongle - Malicious Central

All listed hardware dependencies are connected to the remote Evaluation Machine.

2.2. Software Dependencies

The software dependencies are provided in the artifact script *AnchoredSetup/requirements.sh*. Such scripts are intended to be executed under Ubuntu 22.04. However, the main runtime dependencies are listed below:

- Wireshark 4.1.0 (Included with artifact)
- Python3 $\geq$ 3.8.10 (Included with artifact)
- Zephyr Bluetooth Stack Architecture (Included with artifact)
- Espressif IoT Development Framework v5.0.1

2.3. Project Structure

```
AnchoredSetup ..... Source files for VaktBLE Anchored setup (Python3)
PortableSetup ..... Source files for VaktBLE Portable setup (C++)
  firmware_peripheral ..... Firmware of peripheral side of VaktBLE bridge
  firmware_central ..... Firmware of central side of VaktBLE bridge
Attacks ..... Scripts to Launch BLE attacks (Sweyntooth, CyRC, BLEDiff, InjectaBLE)
firmware_attacker ..... Firmware for Attacker Dongle
firmware_target ..... Sample Firmware for ESP32 Boards (BLE Targets
Eval ..... Python scripts to assist in reproducing results from VaktBLE paper
```

2.4. How to Access

To access the Evaluation Machine through SSH, we can provide our SSH private key (named ***artifact.key***) upon request.

```
chmod 0600 artifact.key
ssh -i artifact.key artifact@evaluation.vaktble.com -p 2222
```

Alternatively, the remote evaluation machine can be accessed using the software AnyDesk (credential can be provided to reviewers upon request).

2.5. Software Installation (Ubuntu 22.04)

The installation of *VaktBLE* is facilitated through the execution of several scripts which handle the installation of all required dependencies for *VaktBLE* itself and other auxiliary scripts to launch BLE attacks and flash the non-compliant *VaktBLE* firmware. The complete installation is described in the steps below:

1. Clone **VaktBLE repository**.

```
cd $HOME
git clone https://anonymous-person@github.com/acsac-2024/vakt-ble-defender.git
```

2. Install system requirements for **Anchored Setup**.

```
cd $HOME/vakt-ble-defender/AnchoredSetup/bridge
./requirements.sh
```

3. Install system requirements for **Portable Setup**.

```
cd $HOME/vakt-ble-defender/PortableSetup
./requirements.sh dev
./build.sh # Build Portable Setup
```

4. Install system requirements for **BLE Attacks**.

This installs the necessary requirements to launch all attacks evaluated in *VaktBLE* paper such as Sweyntooth, BLEDiff, etc.

```
cd $HOME/vaktble-ble-defender/Attacks/
./requirements.sh
```

2.5.1. Firmware Preparation (Optional). The remote machine already contains hardware programmed with all the necessary firmware to run the experiment. However, the instructions on how to flash the firmware is provided below:

1 Flash the ***VaktBLE*** **Firmware**.

With both a nRF52840-Dongle and nRF52840-DK in hand, put the first **nRF52840-Dongle** (non-compliant peripheral) in DFU mode (reset the USB dongle while it is connected to your PC by pressing the small reset button) and run the following command:

```
# Flash the non-compliant peripheral
cd $HOME/vakt-ble-defender/PortableSetup
./firmware.sh peripheral build
./firmware.sh peripheral flash
```

Next, flash Adafruit_nRF52_Bootloader to the **nRF52840-DK** (non-compliant central) following instructions provided in the official project repository. Then, put the nRF52840-DK in DFU mode and run the follow commands:

```
# Flash the non-compliant central
./firmware.sh central build
./firmware.sh central flash
```

2 Flash the **Attacker Firmware**.

Similar to the intructions on how to flash the *VaktBLE* firmware, put the nRF52840-Dongle in DFU mode and run the following commands:

```
# Flash the attacker dongle
cd $HOME/vakt-ble-defender/firmware_attacker
./firmware.py attacker build
./firmware.py attacker flash
```

3 Flash the **InjectaBLE Firmware**.

Similar to the instructions on how to flash the *VaktBLE* firmware, put the nRF52840-Dongle in DFU mode and run the following commands:

```
# Flash the InjectaBLE dongle
cd $HOME/vakt-ble-defender/Attacks/Non-Sweyntooth/injectable/injectable_firmware
make send
```

After the make command finishes, you can verify if the dongle is detected by your operating system. An example of the output is shown in Figure 2..

```
lsusb | grep ButteRFly
```

```
asset@minipc-6:~/vakt-ble-defender/Attacks$ lsusb | grep ButteRFly
Bus 001 Device 043: ID 5a17:0000 Mirage Toolkit ButteRFly
```

Figure 2: OS detecting dongle

4 Flash the **BLE Target Firmware (ESP32-DevKitC)**.

We used a BLE peripheral firmware example from Espressiff project, more precisely *nimble/bleprph* 2023 commit ID: a4afa44. The following commands can download and install the peripheral sample code in a development board with ESP32/ESP32-C3 SoC (e.g., ESP32-DevKitC, ESP-WROVER-KIT,):

```
cd $HOME
git clone https://github.com/espressif/esp-idf.git
cd esp-idf
git checkout a4afa44
./install.sh
source ./export.sh
cd $HOME/vakt-ble-defender/firmware_target/
idf.py set-target esp32
idf.py build # Finally to build and flash
idf.py -p /dev/ttyUSB0 flash
```

3. Evaluation of effectiveness of *VaktBLE* in stopping attacks (RQ1)

Our objective is to replicate the effectiveness of *VaktBLE* in preventing attacks, as presented in Table 2 of the paper. In this section, we provide details of the steps we followed to obtain the data in Table 2. In summary, we manually tested each attack 10 times. The attacks can be found in the ***Attacks/Sweyntooth*** and ***Attacks/Non-Sweyntooth*** directories, respectively. During evaluation, *VaktBLE* outputs a verdict of whether a specific packet belongs to a certain "Validation Type" in the terminal. The list of possible validation types (which corresponds to column "Validation Type" of Table 2 in *VaktBLE* paper) is listed below:

- **Valid** - Indicates that the received packet from the central is valid and **shall be forwarded** to the peripehral under protection (e.g., ESP32/Moto G 5S).
- **Malformed** - Indicates that the packet has been rejected by the ***Decoding*** or ***Filtering*** component (c.f., Figure 4 of *VaktBLE* paper). The packet **will not be forwarded** to the peripheral under protection.
- **Flooding** - Indicates that the packet has been rejected by the ***Filtering*** component. The packet **will not be forwarded** to the peripheral under protection.
- **Out-of-Order** - Indicates that the packet has been rejected by the ***FSM Check*** component. The packet **will not be forwarded** to the peripheral under protection.
- **MIC Error** - Indicates that the packet has been rejected by the ***Encryption*** component. The packet **will not be forwarded** to the peripheral under protection.

3.1. Running *VaktBLE* (Anchored Setup)

Figure 1 illustrates the relevant hardware setup connected to the Evaluation Machine. Once connected via Anydesk or building the code from source with the necessary hardware, the VaktBLE setup can be evaluated. For instance, the BLE target (i.e., Esp32DevKit) attached to the Evaluation Machine has the MAC addess: *c8:c9:a3:d3:65*. Consequently, we can create the benevolent MiTM bridge as follows to test the environment in a **separate terminal**:

```
# Address of target peripheral to protect (ESP32DevKit)
cd $HOME/vaktble-ble-defender/AnchoredSetup/bridge
./run.sh C8:C9:A3:D3:65:1E # Run VaktBLE to defend ESP32DevKit
```

```
Dongles discovered: ['/dev/ttyACM3', '/dev/ttyACM1', 0, 0]
|--> START TX/RX Peripheral Thread --|
[!] reset_bridge_peripheral
[!] reset_bridge_central
[!] Peripheral Address: c8:c9:a3:d3:65:1e, Type: Random (1)
[Valid]    [C --> P | C --- P] RX ---> BTLE_ADV / BTLE_SCAN_REQ
```

Figure 3: Example output for AnchoredSetup

VaktBLE must be running in a separate terminal during all times when launching any BLE attack.

3.2. Launching SweynTooth Attacks

Execution of Sweyntooth attacks is done via a Python 3 virtual environment. Before executing any Sweyntooth attack, such environment has to be activated by running:

```
cd $HOME/vaktble-ble-defender/Attacks/Sweyntooth
source venv/bin/activate
```

Then, you can initiate the Sweyntooth attacks (corresponding to **Table 2** from the paper) by running the following commands (Run 10 times each):

```
export ADDR=c8:c9:a3:d3:65:1e # Set target BDAddress here (ESP32)
./link_layer_length_overflow.py $ADDR # CVE-2019-16336 - Link Layer Length Overflow
./Microchip_invalid_lcap_fragment.py $ADDR # CVE-2019-19195 - Invalid L2cap fragment
./llid_deadlock.py $ADDR # CVE-2019-17060 - LLID Deadlock
./Microchip_invalid_lcap_fragment.py $ADDR # CVE-2019-17517 - Truncated L2CAP
./CC_connection_req_crash.py $ADDR # CVE-2019-19193 - Invalid Connection Request
./sequential_att_deadlock.py $ADDR # CVE-2019-19192 - Sequential ATT Deadlock
./Telink_key_size_overflow.py $ADDR # CVE-2019-19196 - Key Size Overflow
./Telink_zero_ltk_installation.py $ADDR # CVE-2019-19194 - Zero LTKInstallation
```

```
./non_compliance_dhskip.py $ADDR # CVE-2020-13593 - DHCheck Skip
./esp32_hci_desync.py $ADDR # CVE-2020-13595 - ESP32 HCI Desync
./zephyr_invalid_sequence.py $ADDR # CVE-2020-10061 - Zephyr Invalid Sequence
./invalid_channel_map.py $ADDR # CVE-2020-10069 - Invalid Channel Map
```

An example of the output generated by the *AnchoredSetup* is illustrated in Figure 4.

```
[!] Peripheral Address: C8:C9:A3:D3:65:1E, Type: Random (1)
[Valid]    [C --> P | C --- P] RX ---> BTLE_ADV / BTLE_SCAN_REQ
b'pjamm'
|--> START TX/RX Central Thread --|
           [C --> P | C --> P] TX ---> BTLE_ADV / BTLE_CONNECT_REQ
b'cjamm'
           [C --> P | C --> P] TX ---> BTLE_DATA / CtrlPDU / LL_VERSION_IND
           [C --- P | C <-- P] RX <--- BTLE_DATA / CtrlPDU / LL_VERSION_IND
           [C <-- P | C --- P] TX <--- BTLE_DATA / CtrlPDU / LL_VERSION_IND
           [C --- P | C <-- P] RX <--- BTLE_DATA / CtrlPDU / LL_SLAVE_FEATURE_REQ
           [C <-- P | C --- P] TX <--- BTLE_DATA / CtrlPDU / LL_SLAVE_FEATURE_REQ
           [C --> P | C --> P] TX ---> BTLE_DATA / CtrlPDU / LL_LENGTH_REQ
           [C --- P | C <-- P] RX <--- BTLE_DATA / CtrlPDU / LL_LENGTH_RSP
           [C --> P | C --> P] TX ---> BTLE_DATA / CtrlPDU / LL_FEATURE_RSP
           [C <-- P | C --- P] TX <--- BTLE_DATA / CtrlPDU / LL_LENGTH_RSP
[Malformed][C --> P | C --- P] RX ---> BTLE_DATA / L2CAP_Hdr / SM_Hdr / SM_Pairing_Request / Raw
Anomaly Detected by BLEDefender. Terminating connection!
[!] reset_bridge_peripheral
[!] reset_bridge_central
|<-- EXIT TX/RX Central Thread --|
[!] reset_bridge_peripheral
[!] reset_bridge_central
```

(a) AnchoredSetup MiTM example of packet detection output

```
(venv) asset@minipc-6:~/vakt-ble-defender/Attacks/Sweyntooth$ ./link_layer_length_overflow.p
y /dev/ttyACM0 C8:C9:A3:D3:65:1E
Serial port: /dev/ttyACM0
Advertiser Address: C8:C9:A3:D3:65:1E
TX ---> BTLE_ADV / BTLE_SCAN_REQ
Waiting advertisements from c8:c9:a3:d3:65:1e
C8:C9:A3:D3:65:1E: BTLE_ADV / BTLE_ADV_IND Detected
TX ---> BTLE_ADV / BTLE_CONNECT_REQ
Slave Connected (L2Cap channel established)
TX ---> BTLE_DATA / CtrlPDU / LL_VERSION_IND
Slave RX <--- BTLE_DATA / CtrlPDU / LL_VERSION_IND
TX ---> BTLE_DATA / CtrlPDU / LL_LENGTH_REQ
Slave RX <--- BTLE_DATA / CtrlPDU / LL_SLAVE_FEATURE_REQ
TX ---> BTLE_DATA / CtrlPDU / LL_FEATURE_RSP
Slave RX <--- BTLE_DATA / CtrlPDU / LL_LENGTH_RSP
Sending oversized pairing request
TX ---> BTLE_DATA / L2CAP_Hdr / SM_Hdr / SM_Pairing_Request
Connection reset, malformed packet was sent
Waiting advertisements from c8:c9:a3:d3:65:1e
TX ---> BTLE_ADV / BTLE_SCAN_REQ
```

(b) Attacker log generated when sending an invalid SMP_Pairing_Request

Figure 4: (a) Packet detection output from AnchoredSetup MiTM. (b) Log of an attack using an invalid SMP_Pairing_Request.

In the example of Figure 4, the Sweyntooth attack *link_layer_overflow.py* was not successful because *VaktBLE* detected the malformed packet and terminated the connection. This corroborates with the expected result (Validation Type) as reported in Table 2 of *VaktBLE* paper.

3.3. Running Non-Sweyntooth Attacks

For our evaluation described in Section 5 of the paper, we choose state-of-the-art BLE PoCs offensive tools, under ***Attacks/InjectaBLE***, ***Attacks/BLEDiff*** and ***Attacks/CyRC*** directories.
Similarly as Sweyntooth attacks, we executed 10 times each attacks corresponding to Table 2. It is important to note that since KNOB is a Non-Sweyntooth attack, we evaluated the BLE-KNOB Variant provided by Sweyntooth. Using the script under ***Attacks/Sweyntooth/extras/knob_tester_ble.py***. You can execute the command similarly as discussed in section 3.2

3.3.1. Launching InjectaBLE attack. We provide a script to execute such attack. For instance, InjectaBLE allows the performance of central hijacking attacks by injecting a *LL_CONNECTION_UPDATE_IND* packet and synchronizing with the peripheral when the Slave changes its connection parameters. Since this attack requires an established connection, we used Moto G 5S as the central and the ESP32-DevKitC as the peripheral. You can use the the **scrcpy** project to display the screen of the phone in Linux (tested in Ubuntu 22.04) as follows:

```
sudo apt install -y scrcpy
adb devices # Accept ADB authorization from phone
scrcpy # Screen of the phone should open in ~2 seconds
```

In a new terminal you can launch InjectaBLE as follows:

```
cd vakt-ble-defender/Attacks/Non-Sweyntooth/injectable
./attack_hijack_master.sh
```

Once the script starts, the output should look similar as Figure 5.

```
asset@minipc-6:~/vakt-ble-defender/Attacks/Non-Sweyntooth/injectable$ ./attack_hijack_master.sh
[INFO] Module ble_hijack loaded !
[INFO] Module ble_master loaded !
[SUCCESS] ButteRFly device successfully instantiated !
```

Figure 5: Example output for InjectaBLE initialization

Additionally, we installed the nRF Connect Android app on the phone to connect to the target (*nimble-prph*). Once installed, you can open the app and connect via the screen displayed in Ubuntu, as depicted in Figure 6. However, by design, *VaktBLE* rejects this message, preventing InjectaBLE from proceeding further with the attack. Figure 6 shows the log indicating that the attack fails and the process stops.

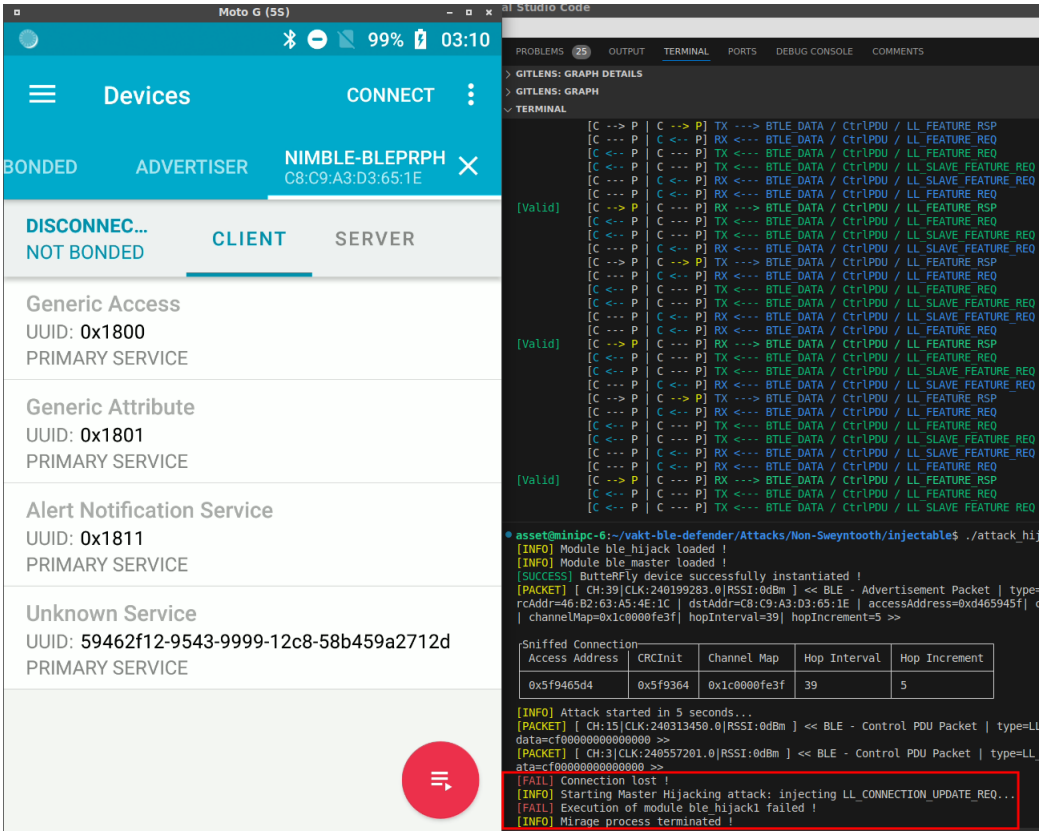


Figure 6: Failed InjectaBLE attack attempt. The attack tries to inject a *LL_CONNECTION_UPDATE_REQ* packet while impersonating the central device. Left side: MotoG 5S already disalready disconnectedto the target device. Upper right: *VaktBLE* logs. Since LL_CONNECTION_UPDATE_REQ is transparent to our bridge, the target device keeps sending *LL_FEATURE_REQ* until timeouts.

3.3.2. Evaluating CyRC. To launch CyRC attacks, we can use the python environment from sweyntooth under ***Attacks/Non-Sweyntooth/CyRC*** directory as follows:

```
cd /vakt-ble-defender/Attacks/Non-Sweyntooth/CyRC
source venv/bin/activate
```

Similarly to the process described in Section 3.2, we can execute each of the commands 10 times to replicate the conditions corresponding to the experiments detailed in Table 2. The following commands can be used to launch the respective attacks:

```
export ADDR=c8:c9:a3:d3:65:1e
#Launch attacks
./assertion_failure_LL_CONNECTION_PARAM.py $ADDR # CVE-2021-3430: Assertion failure on repeated \
    LL_CONNECTION_PARAM_REQ
./assertion_failure_LL_pkts.py  $ADDR # CVE-2021-3433 Assertion failure on certain repeated LL packets
./invalid_channelmap.py $ADDR # CVE-2021-3433: Invalid channel map in CONNECT_IND
./truncated_L2CAP-Kframe.py $ADDR # CVE-2021-3454: L2CAP: Truncated L2CAP K-frame
```

In Figure 7, an example of the attack output is illustrated. In this figure, the attacker on the right side sends multiple *LL_CONNECTION_PARAM_REQ* packets. This sequence of packets triggers the detection of a flooding attack by *VaktBLE*.

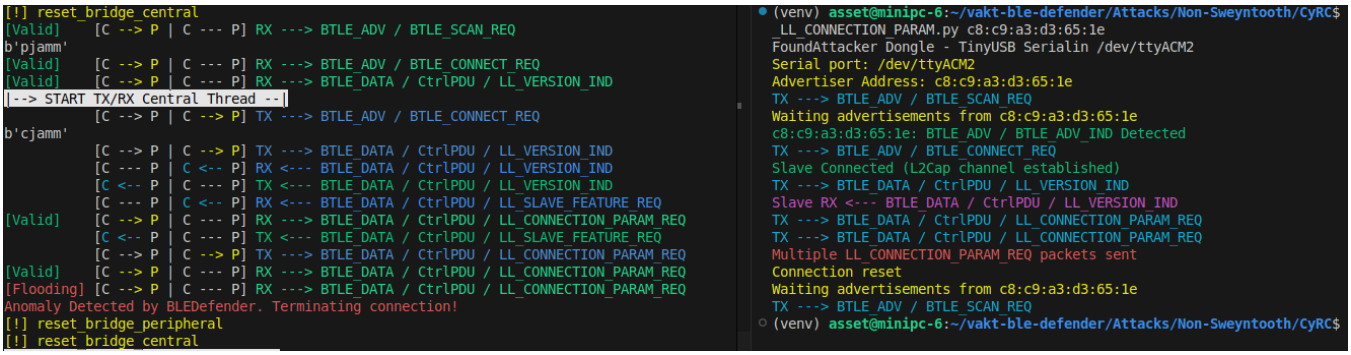


Figure 7: Example output of attacker script aiming to trigger CVE-2021-3430 Assertion failure on repeated LL_CONNECTION_PARAM_REQ

3.3.3. Evaluating BLEDiff. During the evaluation of BLEDiff, we used the MotoG 5S smartphone as the target for two specific attack instances: Bypassing passkey entry in legacy pairing (*E1*) and Bypassing legacy pairing (*E3*) due to legacy pairing compatibility. To enhance our testing efficiency, we modified the attack scripts to automatically detect the *Random Address* of the smartphone. This automation improves reproducibility. Consequently, it is necessary to enable the RANDOM ADDRESS option under the ADVERTISER tab in the nRF Connect app on the smartphone. This step ensures proper communication between the attack scripts and the target device, and is shown in Figure 8.

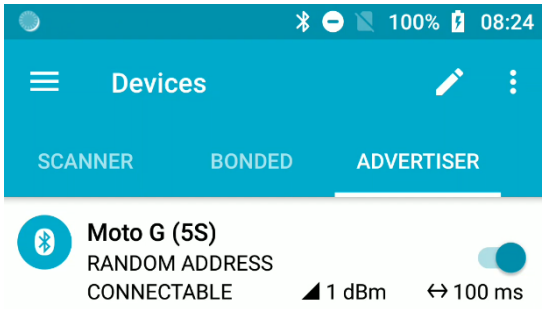


Figure 8: Enabled RANDOM ADDRESS nRF MotoG 5S app

More precisely, we can launch the attacks as follows:

```
./bypassing_legacy_pairing_passkey.py # (E1) Bypassing passkey entry in legacy pairing
./bypassing_legacy_pairing.py# (E3) Bypassing legacy pairing
./key_size_greater_than_max.py c8:c9:a3:d3:65:1e # (O1) Accepting key size greater than max
./e6unresponsiveness_timeout_zero.py c8:c9:a3:d3:65:1e # (E6) Unresponsiveness with Conn. Parameters
```

Figure 9 demonstrates an example of Attack **E1**, showcasing both the attacker's terminal output and the phone screen. This example highlights the attack's behavior when tested on devices supporting legacy pairing, such as the MotoG 5S. It is worth noting that while attacks *E1* and *E3* are specific to devices with legacy pairing support, other attacks like *O1* and *E6* are compatible with a broader range of devices, including ESP32, as detailed in Table 2 of our paper.

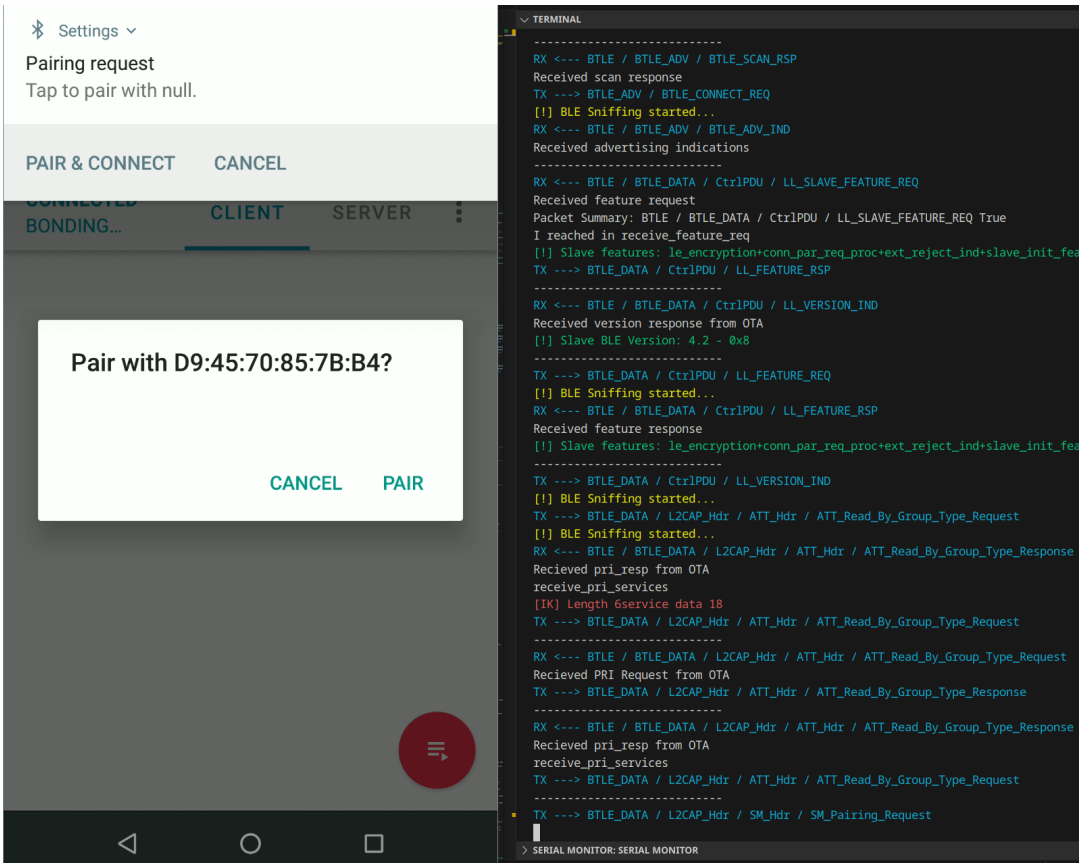


Figure 9: Output from *VaktBLE* and attacker log from launching *bypassing_legacy_pairing_passkey.py* attack

4. Evaluation of efficiency of *VaktBLE* in real-time detection (RQ2)

We evaluated the overhead introduced by *VaktBLE* when forwarding packets during a connection. To this end, we defined a timer l_{start} which is initiated upon identifying a non-malicious packet at MitM peripheral, and l_{end} indicates the time when the packet is forwarded from MitM central, resulting in an overhead $\Delta_t = |l_{end} - l_{start}|$. To compute Δ_t, we added these timers in the *bridge_peripheral_thread()* and *bridge_central_thread()* functions respectively in our main Python code under the file ***/AnchoredSetup/bridge/src/BLEDefender.py***.

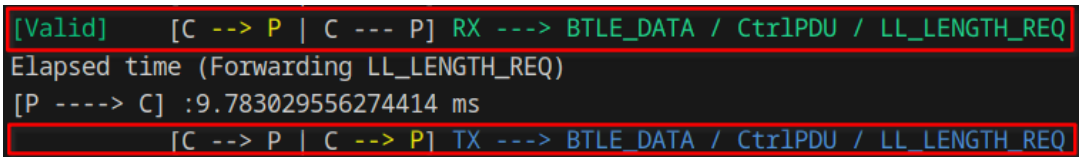


Figure 10: Δ_t time printed by *VaktBLE* when forwarding a packet (i.e., *LL_LENGTH_REQ*

Furthermore, Figure 10 shows an example of the logs of *VaktBLE* printing the elapsed time (Δ_t) when forwarding a packet. To this end, we executed manually 10 times each attack as discussed in section 3.2 and 3.3 , having a total of 250 attacks. However, attacks marked with "N.A." in the *VaktBLE* Overhead (avg) column of Table 2 correspond to attacks within the connection request (i.e., CONNECTION_INDICATION) packet, which cause no measurable *VaktBLE* overhead. Consequently, data was collected for only 200 attack attempts. The resulting data distribution is stored in the file ***/Eval/gen_plots/latency_data_Anchored.txt***. In order to generate the plot, it is necessary to install the libraries required by Python. You can simply run the *requirements.sh* file as follows:

```
cd vakt-ble-defender/Eval/gen_plots
./requirements.sh
# Plot Figure 10 of the paper
./gen_bridge_latency.py
```

4.1. Overhead of *VaktBLE* Portable Setup

We argue that our portable setup significantly reduces the overhead introduced by our anchored setup. It is worth mentioning that we gathered our overhead data from our *PortableSetup* using only one attack (BLE KNOB Variant CVE-2019-9506) and also used the **grep** command to extract the data from the logs. This data distribution can be found under ***/Eval/gen_plots/latency_data_Portable.txt***. Although not shown in the paper, we can similarly plot the distribution by changing the name of the file in the script *gen_bridge_latency.py*.
Furthermore, we can execute our portable setup connected via SSH within our environment as follows:

```
ssh orangepizero3
cd vaktble/
sudo bin/vaktble --debug-pkt-peripheral --name nimble-bleprph --channel 39
```

The *VaktBLE* portable setup utilizes a light-validation codebase implemented in C++, which offers improved performance and reduced overhead compared to our anchored setup running python codebase. While currently employing a lighter set of rules, this portable version is designed with extensibility in mind. It can be readily expanded to incorporate the same comprehensive set of validation rules present in our anchored setup, thus providing a balance between efficiency and robust security measures running python codebase.
An example of a hijacked connection intercepted by *VaktBLE* via our portable setup is more clearly illustrated in Figure 11. This figure provides a visual representation of the significant performance improvement achieved by our portable solution. More precisely, we can observe a substantial reduction in processing time, transitioning from milliseconds in our Anchored setup to microseconds in the portable version. This dramatic decrease in latency highlights the enhanced efficiency of our portable implementation, potentially enabling real-time anomaly detection and mitigation in resource-constrained environments.

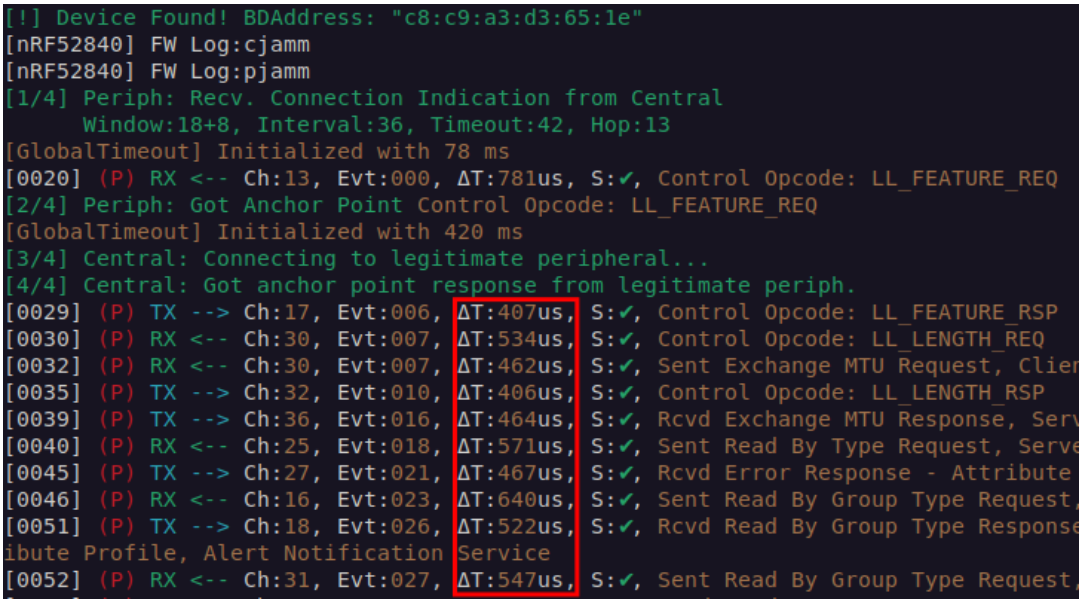


Figure 11: Portable Setup output example, highlighting the overhead introduced (i.e., Δ_t)

5. Evaluation of robustness of *VaktBLE* (RQ3)

Figure 11 of the paper outlines the *VaktBLE* connection successes w.r.t **attacker distance from peripheral**. Since our current hardware setup cannot be physically moved during the evaluation period, reproduction of *VaktBLE* paper robustness results is limited to a single distance between attacker and peripheral. This corresponds to **70cm** (30cm-1m range) as highlighted in Figure 1. Nonetheless, it is possible to evaluate robustness in such fixed distance by launching 10 BLE attacks and checking if the *VaktBLE* terminal output indicates **"Anomaly Detected"**. To this end, the following commands are performed in two separate terminals:

1 Terminal 1 (*VaktBLE*)

```
cd $HOME/vaktble-ble-defender/AnchoredSetup/bridge
./run.sh C8:C9:A3:D3:65:1E
```

2 Terminal 2 (Attacker)

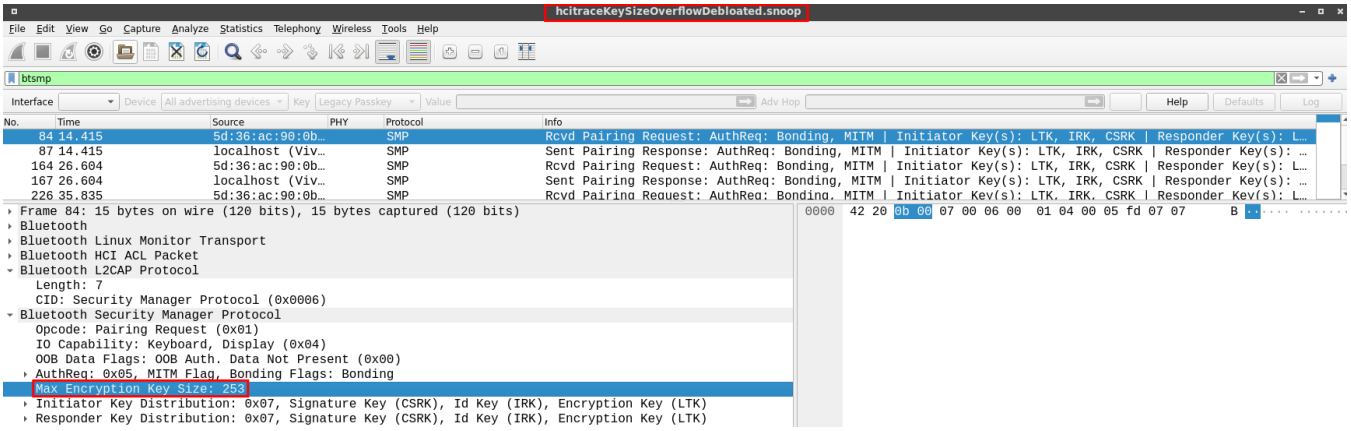
```
export ADDR=c8:c9:a3:d3:65:1e
cd $HOME/vaktble-ble-defender/Attacks/Sweyntooth
source venv/bin/activate
./esp32_hci_desync.py $ADDR
```

The attacker script (**esp32_hci_desync.py**) will start execution and eventually finish. The evaluation of of *VaktBLE* robustness is done by counting the number of attack attempts w.r.t number of connection that *VaktBLE* was able to intercept (i.e., hijack and hence defend the BLE target). The execution of attacks on *terminal 2* is manually performed 10 times for a distance of 70cm between BLE target and attacker. Nonetheless, to facilitate broader evaluation, we are open to move the setup to increase/decrease distance between the attacker and the peripheral. If the evaluation with a different distance is required, kindly reach out to us and we will adjust the physical distance with utmost priority.

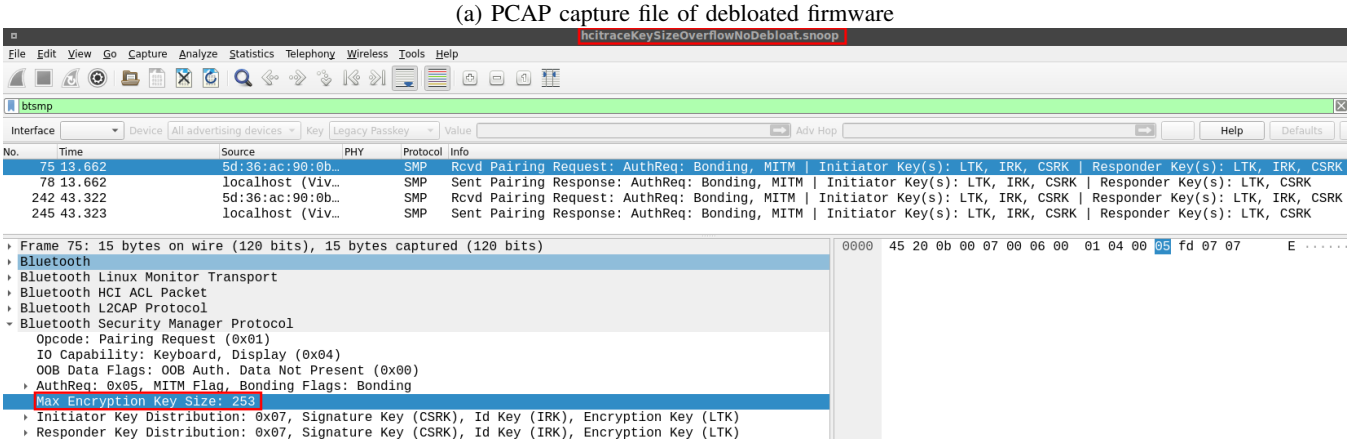
6. Comparing *VaktBLE* against patching-based tools (RQ4)

During this evaluation, logs generated from running LightBLUE, *VaktBLE*, and the sniffer are available in folder ***/vakt-ble-defender/Eval/lightblue*** for each of our evaluated attacks in RQ4 in the paper. However, due to time constraints, the automation of log analysis was not completed, thus no scripts are provided in this. Nonetheless, manual analysis of the logs on different protocols was conducted to confirm that such debloaters are ineffective in protecting against Link Layer attacks.

1) *SMP protocol*: We exemplify capture files such as ***hcitraceKeySizeOverflowDebloated.snoop*** (Figure 12a) and ***hcitraceKeySizeOverflowNoDebloated.snoop*** (Figure 12b). By filtering both capture files using the *btsmp* filter to focus on the SMP protocol, we can observe in Figure 12a that, although debloating is applied, the attack cannot be prevented, as the pairing request is successfully exchanged with an out-of-range value (i.e., 253) according to the BLE core specification.



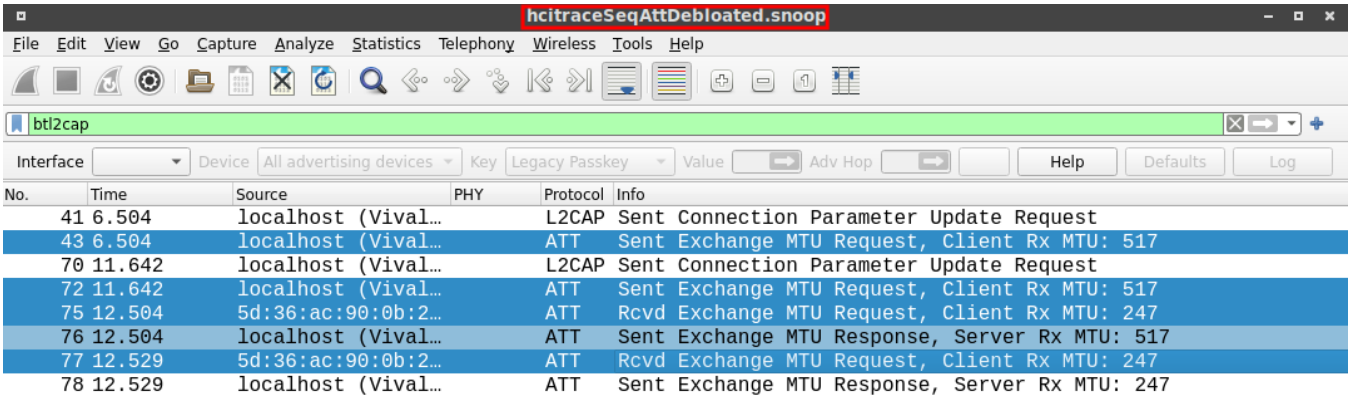
(a) PCAP capture file of debloated firmware



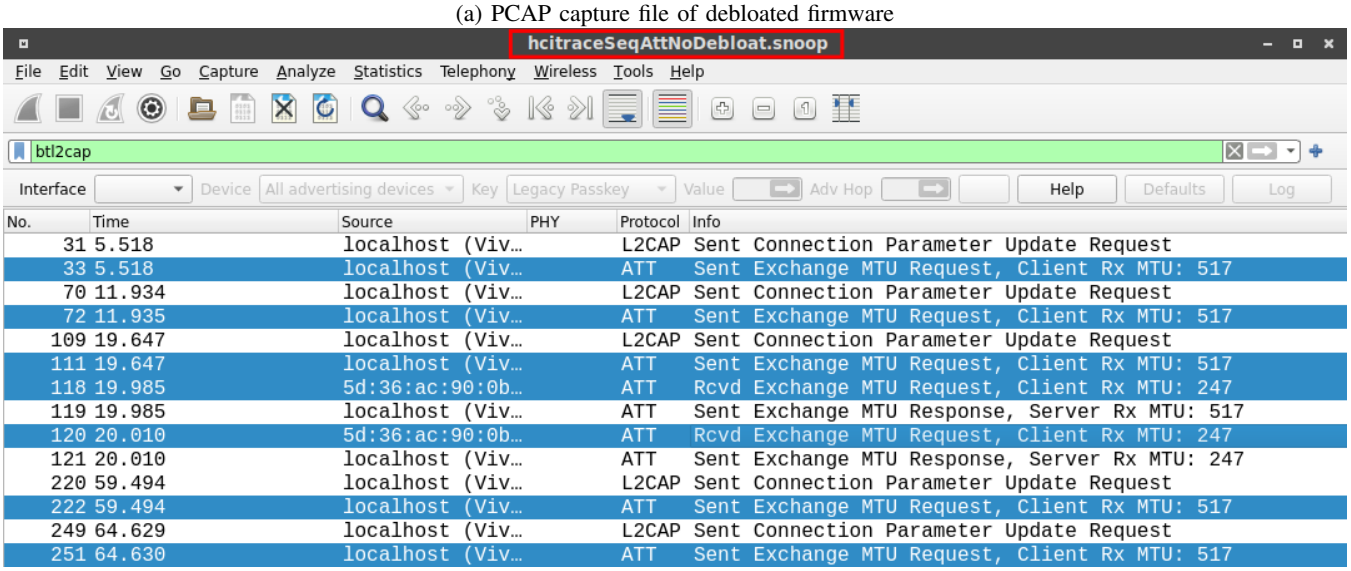
(b) PCAP capture file of non-debloated firmware

Figure 12: Comparison of debloated stack vs Non debloated stack under SMP protocol

2) *L2CAP protocol*: Additionally, we detail our manual analysis for the second layer, specifically the L2CAP (Logical Link Control and Adaptation Protocol). As illustrated in Figures 13a and 13b, multiple MTU Requests are communicated, showing that these requests are consistently sent. This observation underscores that state-of-the-art debloaters are ineffective at protecting against such lower-level attacks, as they fail to address vulnerabilities within the L2CAP layer as well SMP layers.



(a) PCAP capture file of debloated firmware



(b) PCAP capture file of non-debloated firmware

Figure 13: Comparison of debloated stack vs Non debloated stack under L2CAP protocol

7. How effective is *VaktBLE* w.r.t adaptive attacks? (RQ5)

We argue that adaptive attacks are sophisticated methods in which an attacker dynamically adjusts their strategies based on the responses and defenses encountered. To this end, we evaluated four relevant scenarios.

1) **Continuous attacks** (Table 3 in the paper)
2) **Benign and malicious connections** (Table 4 in the paper)
3) **Unknown attacks and/or non-compliant packets** (Figure 12 in the paper)
4) **Different connection parameters** (Table 5 in the paper)

7.1. Continuous attacks

When evaluating *VaktBLE* against continuous attacks, we enhanced our testing methodology by introducing a simple environment variable `CONTINUE_ATTACK` to all scripts. This modification allows for the repeated execution of the same attack multiple times, creating a more sustained attack scenario. Let $e_1, e_2, \ldots, e_n$ represent the set of exploits. With this enhancement, the script launches a continuous attack sequence in the order: $e_1^{10} \rightarrow e_2^{10} \rightarrow \ldots \rightarrow e_n^{10}$, where the superscript 10 indicates that each exploit is repeated 10 times before moving to the next.
To implement this continuous attack scenario, we can set the environment variable in one terminal and repeat one time each command, wait until 10 connection attempts are done and press *Ctrl+C*.

```
cd /vakt-ble-defender/AnchoredSetup/bridge
source ../../Attacks/Sweyntooth/venv/bin/activate
export CONTINUE_ATTACK=1
export ADDR=c8:c9:a3:d3:65:1e # Set target BDAddress here (ESP32)
./link_layer_length_overflow.py $ADDR # CVE-2019-16336 - Link Layer Length Overflow
./Microchip_invalid_lcap_fragment.py $ADDR # CVE-2019-19195 - Invalid L2cap fragment
./llid_deadlock.py $ADDR # CVE-2019-17060 - LLID Deadlock
./Microchip_invalid_lcap_fragment.py $ADDR # CVE-2019-17517 - Truncated L2CAP
./CC_connection_req_crash.py $ADDR # CVE-2019-19193 - Invalid Connection Request
./sequential_att_deadlock.py $ADDR # CVE-2019-19192 - Sequential ATT Deadlock
./Telink_key_size_overflow.py $ADDR # CVE-2019-19196 - Key Size Overflow
./Telink_zero_ltk_installation.py $ADDR # CVE-2019-19194 - Zero LTKInstallation
./non_compliance_dhskip.py $ADDR # CVE-2020-13593 - DHCheck Skip
./esp32_hci_desync.py $ADDR # CVE-2020-13595 - ESP32 HCI Desync
./zephyr_invalid_sequence.py $ADDR # CVE-2020-10061 - Zephyr Invalid Sequence
./invalid_channel_map.py $ADDR # CVE-2020-10069 - Invalid Channel Map
# unset the enviroment variable or just close the terminal
unset CONTINUE_ATTACK
```

7.2. Benign and malicious connections

To evaluate the interplay between benign connections and attacks, we developed a suite of scripts that establish normal connections in accordance with the BLE v5.2 specification. These scripts perform a series of standard operations: establishing a data channel connection, reading the name of the peripheral device, and subsequently terminating the connection. This approach allows us to simulate typical BLE interactions alongside potential attack scenarios. These benign connection scripts are named with the prefix ***fake*** and have a ***.py*** extension. They can be found in the ***/vakt-ble-defender/Attacks/Sweyntooth*** directory. To facilitate testing, we have designed a script that allows for the execution of both benign connections and attack scenarios. These scripts can be run to launch either benign connections or attacks using the following command structure:

```
cd vakt-ble-defender/AnchoredSetup/bridge
./benign_and_attacks.sh
```

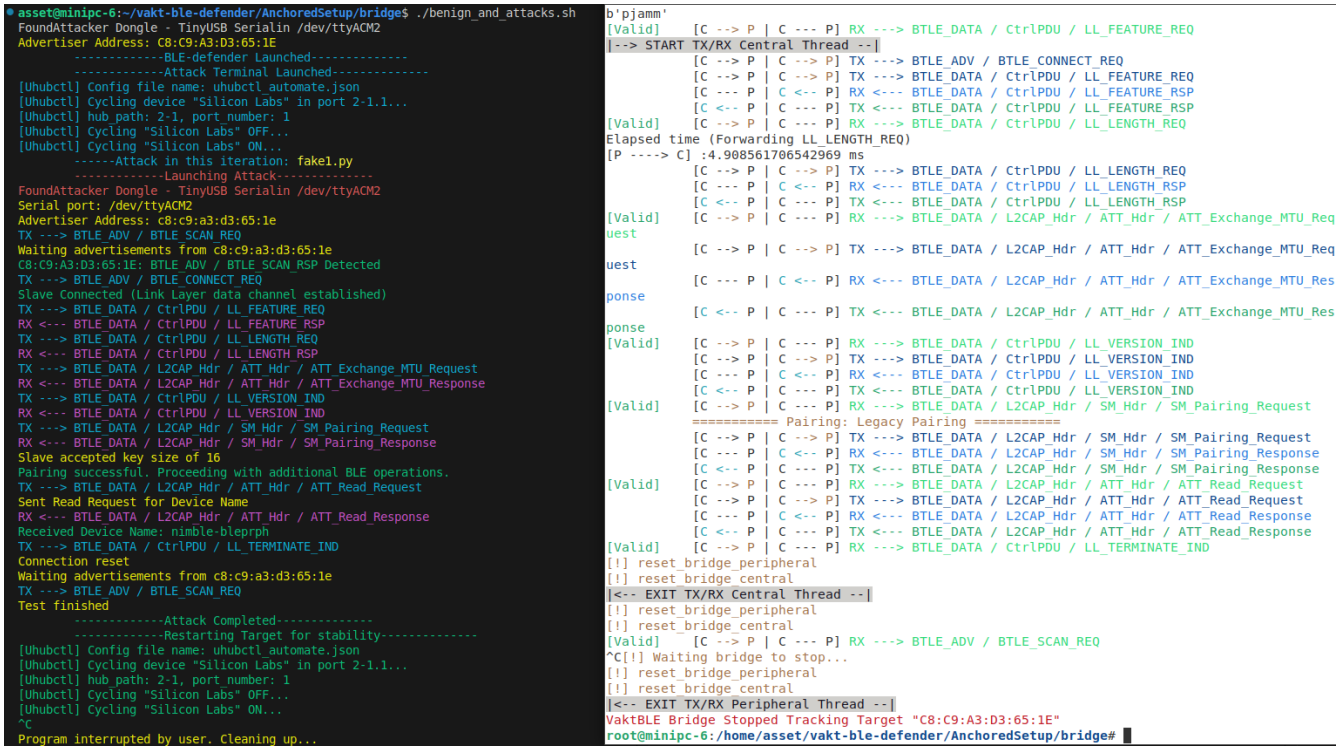


Figure 14: Example output of initialization of benign and attack connection attempts

Our scripts open a new terminal window to run the *VaktBLE* process, as illustrated in Figure 14. The original terminal (shown on the left side of the Figure) maintains the process of launching either attacks or benign connections, based on a binomial distribution probability. This approach allows us to simulate a realistic mix of normal and malicious connections. For our evaluation, as presented in Table 4 of the paper, we allowed the script to run for approximately 10 minutes before terminating it with *Ctrl+C*. This duration provided a sufficient sample size to analyze *VaktBLE*'s performance in detecting and differentiating between benign connections and various types of attacks. The automated nature of this setup enables us to efficiently test *VaktBLE*'s capabilities under diverse conditions, mimicking real-world scenarios where legitimate and malicious activities coexist. By using a binomial distribution to determine the type of connection, we ensure a randomized yet controllable testing environment, enhancing the robustness of our evaluation.

7.3. Unknown attacks and/or non-compliant packets

We aim to demonstrate *VaktBLE*'s robustness against arbitrarily mutated, flooded, and out-of-order packets across more than 25 evaluated attacks. While we recognize that not all fuzzer-generated communications constitute attacks, the majority of the evaluated attacks do originate from such packets. To this end, we leverage Sweyntooth BLE fuzzer (included under ***vakt-ble-defender/Attacks/Fuzzer***).
With the requirements previously installed in Section 2.5, we can carry out the following steps to run the fuzzer:

1) First, we specify the correct parameters for the target in ***vakt-ble-defender/Attacks/Fuzzer/ble_central.py*** at the bottom of the Python code. Specifically, update line 2319 with the target address (*slave_address*) and line 2321 with the dongle serial port from the attacker (*dongle_serial_port*).

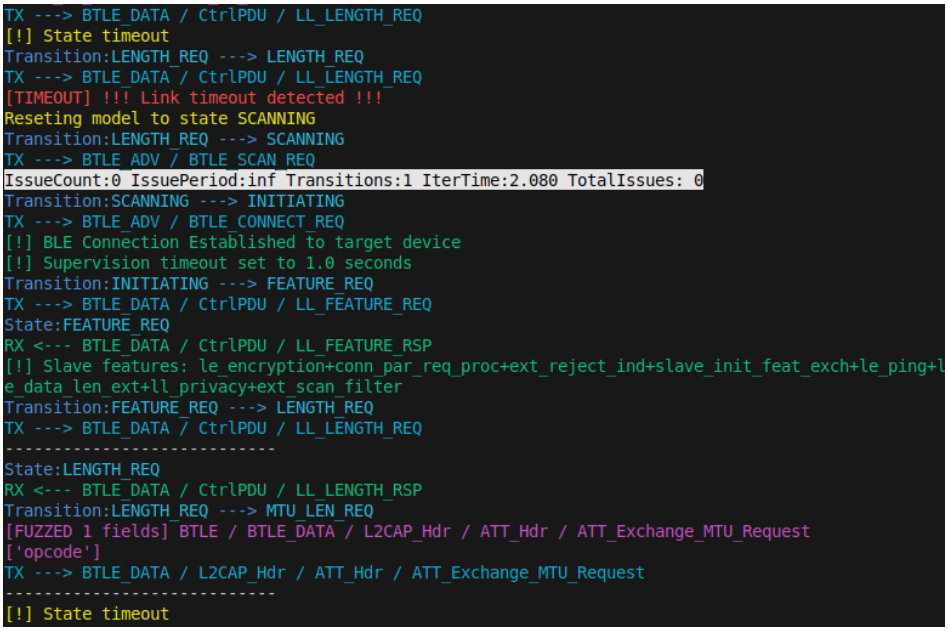
```
model = BLECentralMethods(states, transitions,
                          master_mtu=247,  # 23 default, 247 max (mtu must be 4 less than max \
    length)
                          slave_address=23,
                          master_address='c8:c9:a3:d3:65:1e',
                          dongle_serial_port='/dev/ttyACM2',
                          baudrate=115200,
                          monitor_magic_string='ESP-IDF v4.1'
                          enable_fuzzing=True,
                          enable_duplication=True)
model.get_graph().draw('bluetooth/ble_central.png', prog='dot')
model.sniff()

# try:
while True:
    sleep(1000)
```

2) Secondly we can proceed to execute the fuzzer as follows:

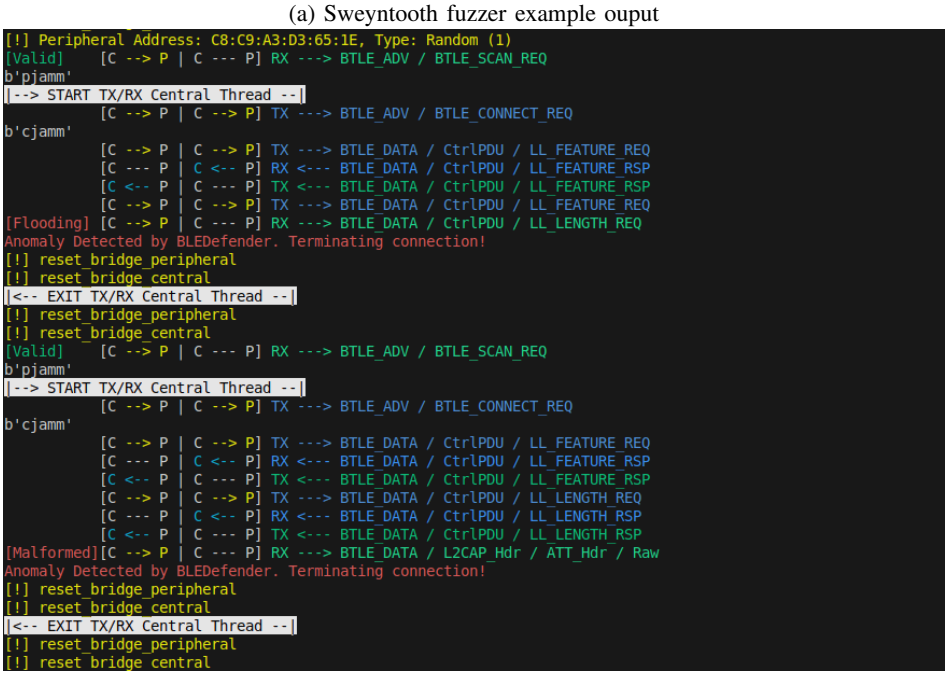
```
sudo ./greyhound.py ble_central
```

An example of the fuzzer logs is illustrated in Figure 15a. While *VaktBLE* logs are highlighted in Figure 15b, for our evaluation, we ran the fuzzer for one hour, splitting the logs into 15-minute intervals. The logs from *VaktBLE* and the SweynTooth fuzzer can be found in ***/vakt-ble-defender/Eval/gen_plots/Fuzzer1hrMutationAndDupEnabled/***. We filtered these logs using *grep* to count the detected anomalies, total number of connection attempts by the fuzzer (ground truth), as well as occurrences of flooding and out-of-order packets.
Finally, by adding such data to our script, we generate Figure 12 in the paper. To this end, simply run the following commands:

```
cd /vakt-ble-defender/Eval/gen_plots
python3 fuzzing_horizontal_stack.py
```



(a) Sweyntooth fuzzer example ouput



(b) *VaktBLE* example log detection of non-compliant packets

Figure 15: (a) Detection of non compliant packets (i.e., Flooding and Malformed) (b) Log of fuzzer duplicating and mutating packets (i.e., *LL_LENGTH_REQ* and *ATT_Exchange_MTU_Request)* respectively

7.4. Different connection parameters

With the objective of making *VaktBLE* miss some connection due to different link-layer connection parameters, we pick a Sweyntooth attack and repeatedly launch it towards the BLE target (ESP32), albeit with different connection parameters at every new attempt. More specifically, we take attack script ***Attacks/Sweyntooth/extras/knob_tester_ble.py*** and modify lines $158 - 164$ before every new launch to use a different parameter within range of the values showcased in Table 5 of *VaktBLE* paper. This is done 10 times for parameters *hop (Hop Interval), chM (Channel Map),interval (Connection Interval), win_size (Windows Size)*. These parameters can be modified directly from the *BTLE_CONNECT_REQ* packet structure and is exemplified below.

```
conn_request = BTLE() / BTLE_ADV(RxAdd=pkt.TxAdd, TxAdd=0) / BTLE_CONNECT_REQ(
    InitA=master_address,
    AdvA=advertiser_address,
    AA=access_address,  # Access address (any)
    crc_init=0x179a9c,  # CRC init (any)
    win_size=2,  # 2.5 of windows size (anchor connection window size)
    win_offset=1,  # 1.25ms windows offset (anchor connection point)
    interval=16,  # 20ms connection interval
    latency=0,  # Slave latency (any)
    timeout=50,  # Supervision timeout, 500ms (any)
    chM=0x1FFFFFFFFF,  # Any
    hop=5,  # Hop increment (any)
    SCA=0,  # Clock tolerance
```

After modifying the parameters, you can launch the Sweyntooth attack script as per usual:

```
export ADDR=c8:c9:a3:d3:65:1e
cd $HOME/vaktble-ble-defender/Attacks/Sweyntooth
source venv/bin/activate
./extras/knob_tester_ble.py $ADDR
```