BrakTooth: Causing Havoc on Bluetooth Link Manager via Directed Fuzzing

Matheus E. Garbelini, Vaibhav Bedi, and Sudipta Chattopadhyay, *Singapore University of Technology and Design;* Sumei Sun and Ernest Kurniawan, *Institute for Infocomm Research, A*Star*

https://www.usenix.org/conference/usenixsecurity22/presentation/garbelini

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

A Artifact Appendix

A.1 Abstract

This artifact provides binaries and OS scripts to evaluate a Bluetooth over-the-air fuzzer under a PC (x86_64) running Ubuntu 18.04. Due to its over-the-air approach and dependency on Bluetooth target devices, access to a remote machine is provided via SSH with private key. Moreover, we design six experiments to assist in replicating the main results of the paper by generating figures and terminal outputs after the fuzzing campaign ends. The evaluation procedure consists of OS scripts that are either included in the artifact or described in this appendix. The results generated by our experiments will help support the claims that (i) our fuzzer outperforms other state-of-the-art over-the-air BT fuzzer, (ii) that our internal fuzzing components are essential and add to the effectiveness of the fuzzer and (iii) that our fuzzing framework is extensible to other wireless protocols beyond Bluetooth such as Wi-Fi and BLE. Lastly, the artifact also includes exploits to launch against real wireless devices (BT, Wi-Fi and BLE) attached to a remote machine.

A.2 Artifact check-list (meta-information)

Obligatory. Fill in whatever is applicable with some keywords and remove unrelated items.

- **Algorithm:** Braktooth OTA Fuzzing
- **Compilation:** GCC version 7.5.0 for modules compilation, fuzzer binaries provided in artifact, source code upon request.
- **Binary:** bt_fuzzer, wdmapper (included with artifact)
- **Run-time environment:** Ubuntu 18.04, Kernel 5.11.13
- **Hardware:** ESP-WROVER-KIT, ESP32-Ethernet-KIT, Oppo Reno 5G, Raspberry 3B and x86_64 Computer
- **Metrics:** Execution Time, Model Coverage, Number of Crashes, Number of Anomalies
- **Output:** Console, files (.txt, .csv) and graphs.
- **Experiments:** Os scripts and manual steps by the user.
- **How much disk space required (approximately)?:** 4 GB
- **How much time is needed to prepare workflow (approximately)?:** 10 min.
- **How much time is needed to complete experiments (approximately)?:** 30 hours.
- **Publicly available (explicitly provide evolving version reference)?:** https://doi.org/10.5281/zenodo.7023642

A.3 Description

The artifact showcases the capabilities of our systematic directed fuzzing framework that automatically discover implementation bugs in arbitrary Bluetooth Classic (BT) devices. We also showcase the flexibility of our approach, which can be applied to other wireless protocols such as Wi-Fi and BLE.

A.3.1 How to access

The access to the target Evaluation Machine can be done via SSH after the reviewer sends his **SSH public key** to the researchers during the artifact evaluation period.

Once access has been granted, the target Evaluation Machine can be accessed via SSH using linux/macos as follows:

```
ssh artifact@evaluation.braktooth.com -p 2222
```

If the reviewer cannot share his/her public SSH public key, we can send our SSH private key (artifact.key), which can be used to access the Evaluation Machine as follows:

```
chmod 0600 artifact.key
ssh -i artifact.key artifact@evaluation.braktooth.com -p 2\
    222
```

X11 forwarding is recommended to be enabled in the SSH client to visualize pdf figures. Otherwise, figure files can be transferred via SFTP.

To access the remote Evaluation Machine from windows, the software MobaXterm can be used as it has X11 enabled by default.

A.3.2 Hardware dependencies

The following hardware development boards are required to evaluate the fuzzer:

- ESP32-WROVER-KIT - Bluetooth Fuzzing Interface
- ESP32-Ethernet-KIT - Vulnerable Bluetooth/Wi-Fi Target
- Oppo Reno 5G - Vulnerable Bluetooth Target
- Raspberry 3B - Vulnerable Wi-Fi Target

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

A.3.3 Software dependencies

The software dependencies for the fuzzer runtime is provided in the artifact script *requirements.sh*. Such script is intended to be executed under Ubuntu 18.04. However, the main runtime dependencies are listed below:

- Wireshark 3.7.0 (Included with artifact)
- Python3 $\geq$ 3.6.9 (Included with artifact)
- Node.js v12.22.12

Furthermore, the vulnerable SDK (esp-idf commit 3de8b79) for the vulnerable target (ESP32) must be installed in the host pc to flash the vulnerable firmware to the target.

Lastly, for comparing our fuzzer with other BT OTA fuzzers, the following 3rd party software is required:

- Bluetooth Stack Smasher v0.6
- BlueFuzz
- bfuzz (iotcube) v2.2.0

Note that the above BT fuzzers are installed via their respective *modules/eval/experimen3-*.sh* scripts.

A.3.4 Data Sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

To avoid causing unintended malfunctions to arbitrary Bluetooth devices, the artifact must be only used against devices which are strictly authorized by the device's owner. Therefore, it is advisable to only fuzz the wireless targets discussed in the experiment workflow.

Furthermore, our remote Evaluation Machine is configured to not log any SSH connection to ensure privacy of the reviewer during the artifact evaluation period.

A.4 Installation

The installation of the fuzzer can be done by running few scripts provided in the artifact binary repository. Download and installation of the fuzzer and software dependencies can be done as follows:

```
mkdir braktooth
cd braktooth
wget https://zenodo.org/record/7023642/files/release.zip
unzip release.zip
# Install software requirements for Ubuntu 18.04
sudo apt install zstd
tar -I zstd -xf wdissector.tar.zst # Extract binary folder
cd wdissector
sudo ./requirements.sh
sudo ./requirements.sh doc # Ignore errors
```

Next, a vulnerable SDK version of our Bluetooth target (ESP32 esp-idf commit 3de8b79) needs to be installed. The following commands can download and install the vulnerable SDK on the remote machine:

```
git clone https://github.com/espressif/esp-idf
cd esp-idf
git checkout 3de8b79 # Vulnerable version of esp-idf SDK
./install.sh
cd ../
```

A.5 Experiment workflow

Figure 1 illustrates the relevant hardware setup in which the experiments are performed on the remote Evaluation Machine. In the following, we describe the experiment workflow which leverages our hardware setup. Note that the fuzzer relies on the BT Fuzzing Interface, which uses the same board model as our *Target 1*, but they are separate boards as illustrated in Figure 1.

First, we aim to evaluate the Bluetooth fuzzer by running it against *Target 1: ESP32-WROVER*. After the maximum number of fuzzing iterations is reached, the fuzzer will stop and generate log files which are used to analyze the results during the experiments via the python scripts provided in the artifact folder *modules/eval*.

As illustrated in Figure 2 (a), the workflow is designed to first evaluate the different variants of the fuzzer by changing its configuration parameters. After each variant is evaluated, the log folder

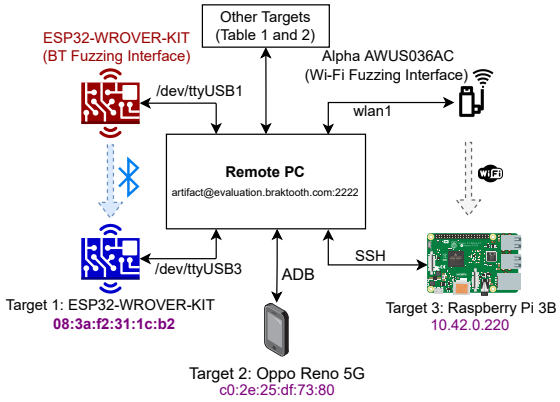

Figure 1: Main Hardware Setup of Evaluation Machine

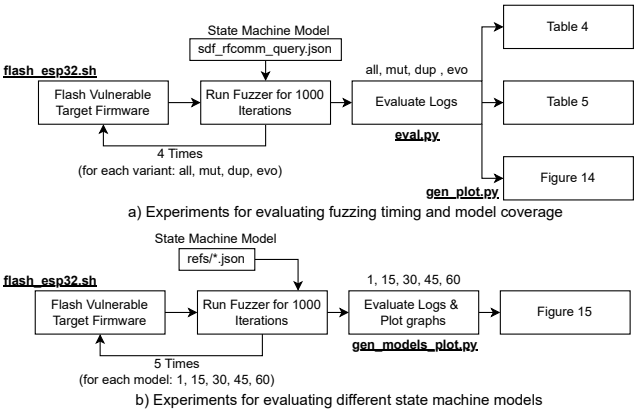

Figure 2: Diagram of Experiments Workflow

Table 1: All BT Devices Setup of Evaluation Machine

BT SoC Vendor	BT SoC	Dev. Kit / Product	BDAddress	Monitor
Bluetooth 5.2				
Intel	AX200	PCIe Module	6C:6A:77:53:97:2D	SSH artifact@127.0.0.1
Qualcomm	WCN399X	Oppo Reno 5G	C0:2E:25:DF:73:80	ADB 627ff0eb
Bluetooth 5.1				
Texas Instruments	CC2564C	CC256XCQFN-EM	98:5D:AD:12:03:F3	Serial /dev/ttyACM0
Bluetooth 5.0				
Cypress	CYW20735B1	CYW920735Q60EVB-01	20:73:5B:1C:D9:93	Serial /dev/ttyUSB6
Bluetrum Technology	AB5301A	AB32VG1	-	-
Zhuhai Jieli Technology	AC6925C	XY-WRBT Module	48:1B:B2:26:90:36	N.A
Actions Technology	ATS281X	Xiaomi MDZ-36-DB	-	-
Bluetooth 4.2				
Zhuhai Jieli Technology	AC6905X	BT Audio Receiver	1F:F6:F7:96:12:E3	N.A
Espressif Systems	ESP32	ESP-WROVER-KIT	08:3A:F2:31:1C:B2	Serial /dev/ttyUSB3
Bluetooth 4.1				
Harman International	JX25X	JBL TUNE500BT	-	-
Bluetooth 4.0				
Qualcomm	CSR 8811	Laird DVK-BT900-SA	-	-
Beken	BK3260N	HC-05	98:DA:60:00:42:E3	Serial /dev/ttyUSB10
Bluetooth 3.0 + HS				
Silabs	WT32i	DKWT32I-A	00:07:80:CC:7D:E3	Serial /dev/ttyUSB4

is fed to the script *eval.py*, which analyzes the packet trace file (*capture_bluetooth.pcapng*) and events logs (events.*.txt) to return timing, coverage and evaluation summary as depicted in Table 4 (Timing of 1000 fuzzing iterations for each device) and Table 5 (Evaluation summary w.r.t. different R_{sel} and D_T) of the paper.

Furthermore, the target is re-flashed via Espressif esp-idf SDK after each re-evaluation. This is to ensure a fresh state to the target in case its flash memory is corrupted during the fuzzing process. Next, after all the fuzzer variants are evaluated (*mutation*, *duplication evolution* and *all*), a graph similar to Figure 14 (crashes/deadlocks w.r.t ESP32 fuzzing iterations) is generated by running the script *gen_plot.py*.

Next, in Figure 2 (b), we illustrate the experiments to evaluate the state machine model of the fuzzer. Similar to the previous experiment, the fuzzer is evaluated multiple times, but with different state machine models. Then, the log results of each evaluation are analyzed by the python script *gen_models_plot.py* and a graph similar to Figure 15 in the paper is generated (Evaluation of different state machine models).

Further, Table 1, lists the BT devices attached to the Evaluation Machine, which can be used to evaluate the bugs and timing behavior depicted in Table 2 and Table 4 of the paper. The column *BDAddress* lists the target Bluetooth Address for fuzzing or exploitation, whereas column *Monitor* describes the monitor connection method with the target such as *Serial*, *SSH* or *ADB*. Moreover, Table 1 corresponds to Table 1 of the paper. However, four devices are currently not possible to remotely evaluate due to the following reasons:

- **Xiaomi MDZ-36-DB and JBL TUNE500BT** - Both BT products (Speaker and headphone respectively) turn off automatically when not receiving any BT connections and requires manual interaction to turn them on before a fuzzing session. Therefore, such devices are not possible to automate for the artifact.
- **Bluetrum AB5301A** - Such board has been updated with the latest proprietary firmware from vendor during the disclosure period, however, we have no copies of the older vulnerable firmware to ensure a proper evaluation with such device. We have contacted the vendor to acquire the older firmware, but we have not received a response so far.
- **Laird DVK-BT900-SA** - The board is non-functional due to a short circuit in the evaluation board which prevent further evaluation. Unfortunately, the DVK-BT900-SA development kit is out of stock as of the time of writing.

Note that the *Mic.* Monitor from the paper is indicated as *N.A* (not applicable) in Table 1 for devices XY-WRBT and BT Audio Receiver since the Evaluation Machine laboratory is in a noisy environment and outside our control. Therefore, the *"Microphone"* monitor is not evaluated in the artifact.

Finally, Table 2 lists the Wi-Fi and BLE devices connected to the Evaluation Machine. Column *"Address"* lists the BLE Address of each device, whereas it is *N.A* for Wi-Fi devices since they connect to the Wi-Fi AP fuzzer automatically. Such Table can be used to replicating Table 7 of the paper (*Summary of unknown flaws found by extension*).

Table 2: Wi-Fi / BLE Devices Setup of Evaluation Machine

Extension	Target	Address	Monitor
BLE Host	ESP32	08:3A:F2:31:1C:B2	Serial /dev/ttyUSB3
	Telink TLSR8258	A4:C1:38:D8:AD:A9	N.A
	NXP KW41Z	00:60:37:88:16:0C	Serial /dev/ttyACM2
	TI CC2540	38:81:D7:3D:45:A2	N.A
Wi-Fi AP	ESP32	N.A	Serial /dev/ttyUSB7
	ESP8266	N.A	Serial /dev/ttyUSB9
	Rasp. Pi 3 B	N.A	SSH pi@10.42.0.220
	One Plus 5T	N.A	ADB 3ffd4d9a

A.6 Evaluation and expected results

The results generated by our experiments will help support the claims that (i) our fuzzer outperforms other state-of-the-art over-the-air BT fuzzer, (ii) that our internal fuzzing components are essential and add to the effectiveness of the fuzzer and (iii) that our fuzzing framework is extensible to other wireless protocols beyond Bluetooth such as Wi-Fi and BLE.

Evaluation Instructions:

We start by flashing a vulnerable firmware into the target esp32 which is connected to the remote machine. A code snipped of the procedure is shown in Listing 1. For your convenience, such script is included in *modules/eval/flash_esp32.sh*.

Listing 1: Flashing firmware to ESP32 target (flash_esp32.sh)

```
cd esp-idf
source export.sh
cd examples/bluetooth/bluedroid/classic_bt/bt_spp_acceptor
idf.py build
# Program firmware to target (connected via /dev/ttyUSB3)
idf.py -p /dev/ttyUSB3 erase_flash flash
```

A.6.1 Experiment 1 - Evaluating Timing, Coverage and Fuzzing Components

This experiment is intended to run the fuzzer in different configurations to evaluate the components that contribute to the overall design of the fuzzer. The script included on *modules/eval/experiment1.sh* runs the fuzzer 4 times, switching between the fuzzing parameters *–mutation, –duplication, –optimization*.

The script below can be used to run experiment 1 for a ESP32 target with BDAddress of 10:52:1c:69:ac:82.

```
cd $HOME/braktooth/wdissector/modules/eval
./experiment1.sh
```

When running the script above, the terminal output illustrated in Figure 3 appears during the fuzzing session, indicating an exchange of over-the-air LMP packets between the fuzzing interface and the target *ESP32* device. Furthermore, upon end of evaluation (which takes several hours), extra logging folders and files are created as illustrated in Figure 4.

Now, we can start to analyze the outputs generated and relate to the tables and figures present in the paper. To start, we can get

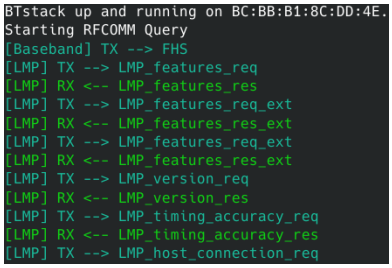

Figure 3: Expected output when the BT fuzzer is running

```
wdissector
└── modules
    └── eval
        ├── all ..... Logs for Variant 'All'
        ├── mut ..... Logs for Variant 'Mutation'
        ├── dup ..... Logs for Variant 'Duplication'
        ├── evo ..... Logs for Variant 'Evolution'
        └── graph_models.pdf .. Figure 14 in the paper
```

Figure 4: Generated folder and files after running *experiment1.sh*

the coverage and timing for the variant 'all', which corresponds to Table 4 in the paper, by running the following command in the *eval* folder:

Listing 2: Generate results for Table 4 (Timing of 1000 fuzzing iterations for each device)

```
cd $HOME/braktooth/wdissector/modules/eval
./eval.py all
```

The output of Listing 2 should look similar to Figure 5, thus returning relevant information that is present in Table 4 of the paper such as *Total Time, 1st Vulnerability, 1st Non-compliance* and *Model Coverage* (highlighted in blue). Although the time to complete 1000 iterations (*Total Time*) can variate, it is usually in the range of 3h-3:30h for a BT target such as ESP32. Nevertheless, *Model Coverage* for ESP32 has its value in the range $22-30\%$ for this evaluation, as exemplified in Figure 5. Moreover, due to the stochastic behavior of the over-the-air fuzzing process, the *1st Vulnerability, 1st Non-compliance* can variate significantly. Depending on the iteration, the first Vulnerability or Non-Compliance can be achieved in less than 1 minute in an optimistic scenario or after dozens of minutes, or almost one hour in worst case.

This experiment to generate *Table 4* mainly focuses on ESP32. However, customization of the experiment for evaluation of other BT targets is discussed in section A.7.1.

Next, the first entry of Table 5 of the paper (Evaluation summary w.r.t. different R_{sel} and D_T) is obtained by running the script of Listing 3 and getting the output of *"Evaluation Summary"* as illustrated in Figure 6. Note that the python script *eval.py* receives "*dup*" as argument, which refers to the *Duplication* fuzzing variant log folder, which was generated after running *experiment1.sh*.

Listing 3: Generate tesults for Table 5 (Evaluation summary w.r.t. different R_{sel} and D_T)

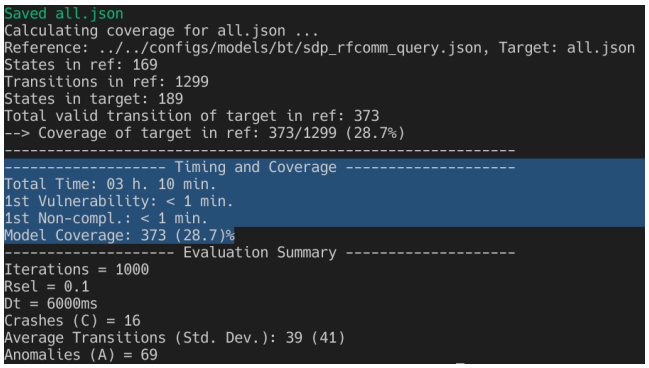

Figure 5: Example output for Table 4 results (ESP32 target)

```
cd $HOME/braktooth/wdissector/modules/eval
./eval.py dup
```

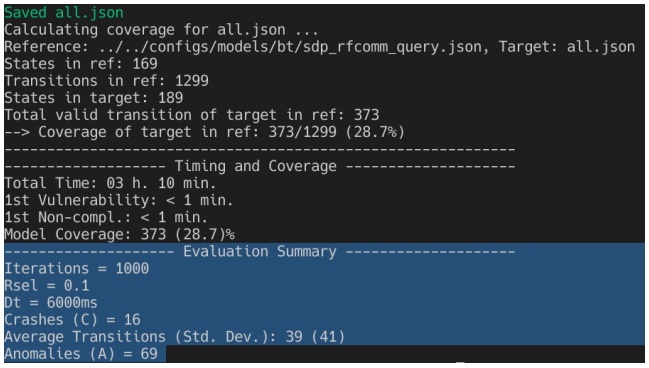

Figure 6: Example output for an entry of Table 5

It is worthwhile to mention that the generated result for this experiment is based on 1000 iterations instead of 200 iterations as described on the paper to avoid running an additional evaluation. Furthermore, since it requires a total of 9 evaluation to generate all entries of Table 5 as shown in the paper, this experiment only focuses on the first one to save time during this experiment. Nevertheless, generating more entries or limiting the number of iteration can be done by changing certain configuration parameters as later discussed in the Experiment Customization (Section A.7.1).

Furthermore, the output obtained for Listing 3 shall indicate more *Crashes (C)* than indicated on the paper due to the extended maximum number of iterations, which results in more time to find crashes. On the other hand, the *Average Transitions (Std. Dev.)* should stay relatively the same as indicated on Table 5 in the paper (107 ± 81 for entry $R_{sel} = 0.1$ and $D_T = 6000$).

Lastly, we can generate a figure similar to the Figure 14 presented in the paper (crashes/deadlocks w.r.t ESP32 fuzzing iterations), albeit not for *unique crashes/deadlocks*, but rather for all reported crashes from the fuzzer. This is because our framework cannot automatically detect the root cause of each reported crash. Instead, the uniqueness shown on Figure 14 in the paper, requires manual and careful analysis of the target trace output. Automation of such effort to investigate the root cause is beyond the scope of our fuzzing tool.

Nevertheless, a figure for crashes/deadlocks w.r.t ESP32 fuzzing iterations is generated and opened by running the script below:

```
cd $HOME/braktooth/wdissector/modules/eval
./gen_plot.py
# Graph saved to graph_optimization.pdf.pdf
```

In the case the figure failes to open to your view, you can manually copy the figure locally via SFTP or call *okular* on the remote host machine. The latter approach requires X11 enabled in your SSH client:

```
okular graph_optimization.pdf
```

Figure 7 depicts a sample of the expected graph for this evaluation.

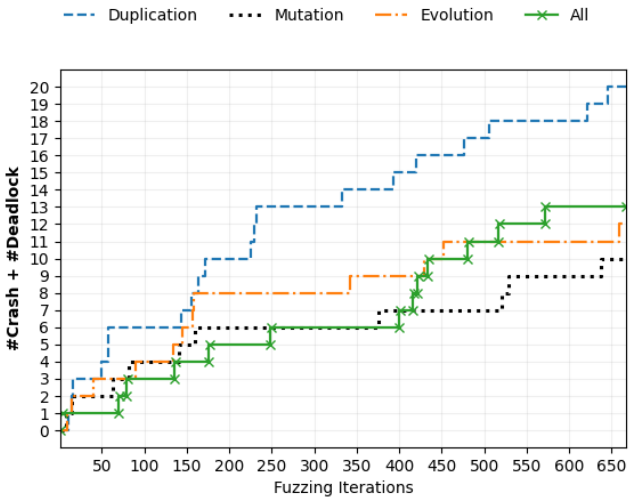

Figure 7: Sample graph for crashes/deadlocks w.r.t ESP32 fuzzing iterations.

A.6.2 Experiment 2 - Evaluating State Machine Model

As illustrated in the diagram of Figure 2 (b), this second experiment focuses in evaluating the differences in coverage, number of crashes and anomalies for different reference models used during the Bluetooth fuzzing session. To this end, the script *modules/eval/experiment2.sh* has been prepared to automate the generation and selection of the models before the evaluation starts. The relevant files used for the model generation are depicted in Figure 8.

This experiment already provides such reference capture to simplify the evaluation, however, Section A.7.2 details how to create clean reference captures that can be used to create reference models.

After running the script of Listing 4, you should get (after several hours) the terminal output depicted in Figure 9 and the evaluation graph of all the reference models as illustrated in Figure 10.

Listing 4: Generate results for Figure 15 (Evaluation of different state machine models)

```
cd $HOME/braktooth/wdissector/modules/eval
./experiment2.sh
# Graph saved to graph_models.pdf
okular graph_models.pdf
```

Figure 9 and Figure 10 relates to Figure 15 of the paper and depicts the number of states, model coverage, number of crashes

```
wdissector
└── modules
    └── eval
        └── refs
            ├── 1.pcapng  ..... 1 min. Reference capture
            ├── 15.pcapng  .... 15 min. Reference capture
            ├── 30.pcapng  .... 30 min. Reference capture
            ├── 45.pcapng  .... 45 min. Reference capture
            └── 60.pcapng  .... 60 min. Reference capture
```

Figure 8: Reference capture files used for reference model generation.

```
-------------------- Graph Summary ----------------------
====== eval_1 ======
Max iterations: 1000
Crashes: 10
Anomalies: 62
Coverage: 80.1
States count: 82
LMP states: 62
Non-LMP states: 20
====== eval_15 ======
Max iterations: 1000
Crashes: 7
Anomalies: 63
Coverage: 61.8
States count: 88
LMP states: 68
Non-LMP states: 20
====== eval_30 ======
Max iterations: 1000
Crashes: 8
Anomalies: 63
Coverage: 59.0
States count: 93
LMP states: 73
Non-LMP states: 20
====== eval_45 ======
Max iterations: 1000
Crashes: 5
Anomalies: 63
Coverage: 55.1
States count: 101
LMP states: 81
Non-LMP states: 20
====== eval_60 ======
Max iterations: 1000
Crashes: 9
Anomalies: 59
Coverage: 48.2
States count: 107
LMP states: 87
Non-LMP states: 20
------------------------------
Figure saved as graph_models.pdf
```

Figure 9: Sample terminal output of evaluation of different state machine models.

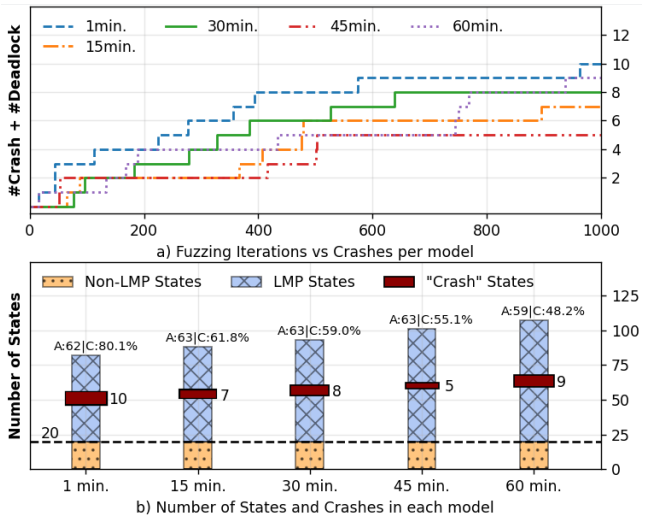

Figure 10: Sample graph of evaluation of different state machine models (graph_models.pdf).

and anomalies for each of the five reference models evaluated as described in the paper ($\{M^i_{ref} \mid i \in \{1, 15, 30, 45, 60\}\}$).

Overall, the number of states should slightly increase according to the model with the highest training time (M^{60}_{ref}) while the coverage should decrease for such model. This is because with a more complete reference model such as M^{60}_{ref}, more states are to be explored during the 1000 fuzzing iterations, which translates to a lower coverage as compared to a simpler reference model such as M^{1}_{ref}.

Customization of this experiment on how to generate reference captures from scratch is discussed in Section A.7.2.

A.6.3 Experiment 3 - State Mapping Generation

The artifact includes several reference capture files from protocols beyond BT Classic to evaluate the state mapper. However, in order to evaluate the state mapper, we can run the script of Listing 5 to generate the complete state machine visualization of the simplified graph presented in Figure 16 of the paper (An illustration of a simplified BT state machine and corresponding state mapping rules for LMP and L2CAP.). The state machine generation takes as input a reference capture (*capture_bt_a2dp.pcapng*) and the configuration file with the mapping rules (*config_bt.json*).

Figure 11 illustrates the generated state machine graph for the sample BT capture (*capture_bt_a2dp.pcapng*) and should correspond to Figure 16 of the paper.

Listing 5: Run state mapper for sample capture files (Figure 16 of the paper).

```
cd $HOME/braktooth/wdissector/examples/wdmapper/
# This will generate states_bt_a2dp.svg
./run_example_wdmapper.sh
sudo npm install svg2pdf -g
svg2pdf states_bt_a2dp.svg # Convert svg to pdf
okular states_bt_a2dp.pdf # Open pdf
```

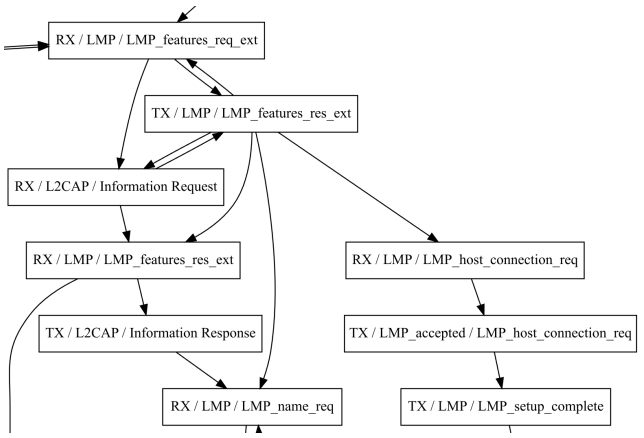

Figure 11: Simplified BT state machine and corresponding state mapping rules for LMP and L2CAP (states_bt_a2dp.svg).

A.6.4 Experiment 4 - Comparison between different fuzzing tools

On this experiment, we evaluate other BT OTA fuzzer against our ESP32 target and compare the results to Table 6 (A Comparison among different fuzzing tools).

For this experiment, we don't validate the entry *toothpicker* since it requires a special hardware setup which is outside the scope of our remote evaluation platform.

Nevertheless, the other third parties fuzzing tools (*bfuzz, Stack Smasher and Bluefuzz*) are installed and executed by running the following scripts:

```
cd $HOME/braktooth/wdissector/modules/eval
./experiment3_bss.sh # Stack Smasher
./experiment3_bfuzz.sh # bfuzz (iotcube)
./experiment3_bluefuzz.sh # Bluefuzz
```

While the first script (*bss*) does not require user interaction, the other scripts (*bfuzz and bluefuzz*) require the user to select the Bluetooth device before starting the fuzzing session. To this end, once such script are executed and a prompt asking for device number is requested, the user needs to select the number for device name "ESP_SPP_ACCEPTOR" and its respective BT service options as illustrated in Figure 12 and Figure 13 for the fuzzers *bfuzz* and *Bluefuzz* respectively.

It is worthwhile to mention that the scripts for this experimented are not completely automated, so the evaluator would need to press the keys *CTRL + C* to interrupt the fuzzing session after 3 hours for each script.

For all scripts, once a crash has been triggered, the terminal output should show the crash indication message for each fuzzer such as "Crash detected". Nevertheless, if the target (ESP32) becomes unresponsive during the session, the experiment script can be re-run to reset the target firmware.

It is expected that only *bfuzz (iotcube)* is able to trigger a crash in ESP32, however due to the random nature of the other fuzzers (*bss, Bluefuzz*), receiving a crash during the 3 hours evaluation period is still possible. However, we claim that our fuzzer outperforms the *state-of-the-art* (at the time of writing the paper) by finding new

bugs and non-compliance in the LMP layer of ESP32.

```
[+] Start scanning bluetooth devices...

        Target Bluetooth Device List
        [No.]   [BT address]            [Device name]
        00      08:3A:F2:31:1C:B2       ESP_SPP_ACCEPTOR
        01      5C:66:6C:6C:AC:98       OPPO Reno2 Z
        02      FC:77:74:9E:00:97       DESKTOP-Q3P21MS
        03      94:65:2D:ED:34:1F       OnePlus 5T
        Total : 4

select device: 00

Start scanning services...

        List of profiles for the device
        00      [0x1101]: Serial Port
        01      [0x0000]: SDP
        Total : 2

Select a profile to fuzz: 00
```

Figure 12: bfuzz BT device options screen.

```
Chose a device number for pairing (q for quit):0
You have chosen device 0: 08:3A:F2:31:1C:B2(ESP_SPP_ACCEPTOR)
Chose the service number :0
protocol: RFCOMM
port:  1
```

Figure 13: Bluefuzz BT device options screen.

Customization of this experiment on running the comparison against other BT devices of Table 1 is discussed in Section A.7.3.

A.6.5 Experiment 5 - Attacks Exploiting BrakTooth

In this experiment, we reproduce certain BT attacks against ESP32 as reported in the paper. We also provide an example of launching an attack against *Oppo Reno 5G* via the SSH monitor included in the fuzzer. Note that in this experiment we use ***Oppo Reno 5G*** instead of *Pocophone F1* as used in the paper due to unavailability of our *Pocophone F1* during the evaluation artifact period. Nevertheless *Oppo Reno 5G* uses the same BT SoC (WCN399X) as *Pocophone F1* and therefore is vulnerable to the same BT attacks.

Before launching the attack, we need to know the BDAddress of the target BT device. To facilitate this, BT Exploiter can scan the BDAddress of targets nearby by running the following command:

```
sudo bin/bt_fuzzer --scan
```

If ESP32 is detected, then you should get a similar output as shown in Figure 14.

```
Serial port /dev/ttyUSB3@4000000 opened
BT Scanning Started (Inquiry)...
[ESP32BT] BDAddress: 94:65:2d:ed:34:1f, Name: OnePlus 5T, RSSI: -54, Class: Smartphone
[ESP32BT] BDAddress: 08:3a:f2:31:1c:b2, Name: ESP_SPP_ACCEPTOR, RSSI: -30, Class: Unkown
[ESP32BT] BDAddress: fc:77:74:9e:00:97, Name: DESKTOP-Q3P21MS, RSSI: -42, Class: Desktop workstation
```

Figure 14: BT Scan output

Next, we can choose an exploit by its name and use the target BDAddress. For this example, evaluation, we start by launching the remote code execution attack against ESP32 as described in the paper (CVE-2021-28138) by running the following command:

```
sudo bin/bt_fuzzer --no-gui --exploit=\
     invalid_feature_page_execution
--target=08:3a:f2:31:1c:b2 --target-port=/dev/ttyUSB3
```

If the attack is successful, the fuzzer output should log the crash trace of the target ESP32 with a program counter (PC) set to 0*xdeadbeee*. Thus, indicating that we have control over the target's program counter.

```
[LMP] TX --> LMP_features_res_ext
[L2CAP] RX <-- Connection Response - Success (SCID: 0x0041, DCID: 0x0040)
[L2CAP] TX --> Configure Request (DCID: 0x0040)
[L2CAP] RX <-- Configure Request (DCID: 0x0041)
[L2CAP] TX --> Configure Response - Success (SCID: 0x0040)
E (656470) BT_BTM: btm_process_remote_ext_features: page=3 unexpected

[Crash] Crash detected at state TX / L2CAP / Configure Response
------------------------------------------------------------
[Optimizer] Iter=3  Params=[0.00472128,0.0751069,0.136429,0.190592,0.0687216,0.15383,...
[Optimizer] Fitness=47  Adj. Fitness=-47
------------------------------------------------------------
51:[LMP] TX --> LMP_detach
Host BDAddress randomized to 2f:57:93:6f:01:9c
[!] Global timeout started with 45 seconds
Guru Meditation Error: Core  0 panic'ed (IllegalInstruction). Exception was unhandled.

Core  0 register dump:
PC      : 0xdeadbeee  PS      : 0x00060031  A0      : 0x80087218  A1      : 0x3ffbe360
A2      : 0x3ffb80ec  A3      : 0x3ffb1a3c  A4      : 0x00000001  A5      : 0x000ffffc
A6      : 0xdeadbeee  A7      : 0x00060b23  A8      : 0x800832cc  A9      : 0x3ffbe340
A10     : 0x3ffb1a3c  A11     : 0x3ffba498  A12     : 0x00000000  A13     : 0x00000001
```

Figure 15: Output of Arbitrary code execution on ESP32 (CVE-2021-28138)

Following page 13 of the paper (DoS in Laptops & Smartphones), we can launch a denial-of-service attack against a smartphone (Oppo Reno 5G) and monitor it via ADB. To this end, change the parameter *"MonitorType"* to **3** in *configs/bt_config.json* (using nano or vim for example) and run the fuzzer with the *"invalid_timing_accuracy"* exploit.

```
 sudo bin/bt_fuzzer --no-gui --exploit=\
invalid_timing_accuracy --target=c0:2e:25:df:73:80 \
--target-port=/dev/ttyUSB3
```

Run the command above for about 2 minutes and stop the fuzzer with *CTRL + C*. Since the output of the phone via logcat is too fast, we need to manually check the target log (*logs/Bluetooth/monitor.1.txt*) to validate if a crash has been triggered on the SoC of the target.

```
cd $HOME/braktooth/wdissector/logs/Bluetooth
cat monitor.1.txt | grep -i "SoC Crashed"
```

If the target BT firmware has crashed and the attack was successful, the output of the command about should return the string ***"Primary Reason for SoC Crash:SOC crashed"***

Finally, the evaluator can optionally launch exploits to trigger the bugs in Table 2 of the paper. This customization is elaborated in Section A.7.4.

A.6.6 Experiment 6 - Fuzzing Extensions

This section evaluates the claim that our fuzzer is extensible to other wireless protocols such as Wi-Fi and BLE Host by running an exploit against ESP32 (BLE Host) and Raspberry Pi 3B (Wi-Fi). We leave the replicability of the "coverage" of Table 7 (Summary of unknown flaws found by extension) on the paper as optional since demonstrating the exploits confirms the extensibility of the fuzzer.

BLE Host Fuzzer: Starting with the BLE host fuzzer, we need to flash a BLE firmware to ESP32 with a slight modification to the sample code "gatt_security_server":

```
nano $HOME/esp-idf/examples/bluetooth/bluedroid/ble/\
     gatt_security_server/main/example_ble_sec_gatts_demo.\
     c
```

```
# Modify the following
-    .own_addr_type      = BLE_ADDR_TYPE_RANDOM,
+    .own_addr_type      = BLE_ADDR_TYPE_PUBLIC,

-            esp_ble_gap_config_local_privacy(true);
+            esp_ble_gap_config_local_privacy(false)
```

After modifying the source code as instructed above. You can build and flash the new firmware to the ESP32 target:

```
$HOME/esp-idf/
source export.sh
cd $HOME/esp-idf/examples/bluetooth/bluedroid/ble/\
    gatt_security_server/
idf.py build
idf.py -p /dev/ttyUSB3 flash_erase flash
```

Now, we can launch the *"Null Dereference"* exploit from the fuzzer as follows:

```
sudo bin/bthost_exploiter --target=08:3a:f2:31:1c:b2 \
    --exploit=esp32_bluedroid_pairing_crash
```

If the null pointer dereference attack (CVE-2022-26604) is successful, you should the output indicated in Figure 16.

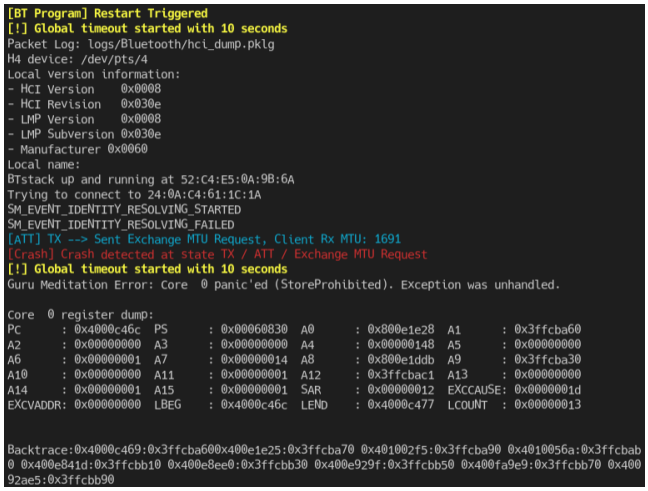

Figure 16: ESP32 Bluedroid Null Pointer dereference (CVE-2022-26604)

Moreover, to run the BLE Host in normal mode (without any exploits) and replicate Table 7, the fuzzer can be launched as follows:

```
sudo bin/bthost_exploiter --target=08:3a:f2:31:1c:b2 \
    --mutation=true
--duplication=true --optimization=true --max-iterations=1\
    000
cp logs/BTHost modules/eval/custom
cd modules/eval
./eval.py custom
```

The customization of this experiment to target other targets as shown in Table 2 is discussed in A.7.5.

Wi-Fi AP Fuzzer:

Next, we repeat the exploitation experiment for the Wi-Fi AP fuzzer by launching the *Probe Resp. Deadlock* (CVE-2022-26599) against Raspberry Pi 3B. To launch such attack, we need to force the Wi-Fi client (Raspberry Pi) to connect to our Wi-Fi AP by running a script that ensures reconnection Wi-Fi reconnection. We provide this Raspberry Pi script in our remote setup via SSH at 10.42.0.220:

Listing 6: Wi-Fi Client reconnection script

```
ssh pi@10.42.0.220 # No password needed
cd WiFiSuite/wifisuite/
sudo dmesg -C && sudo dmesg -w &
sudo python test.py
```

After the previous commands were issued in Raspbery Pi, it will try to connect to an AP matching the name (SSID) "TEST_KRA". Now, to start the Wi-FI AP fuzzer, start a new SSH terminal on the remote machine and run the following:

Listing 7: Wi-Fi Client reconnection script

```
cd $HOME/braktooth/wdissector
sudo bin/wifi_ap --exploit=broadcom_bad_prob_rsp
```

After a couple of minutes (about 1-2 minutes), the attack is successful if the Raspberry Pi SSH terminal shows the string *"firmware has halted or crashed"* (c.f., Figure 17).

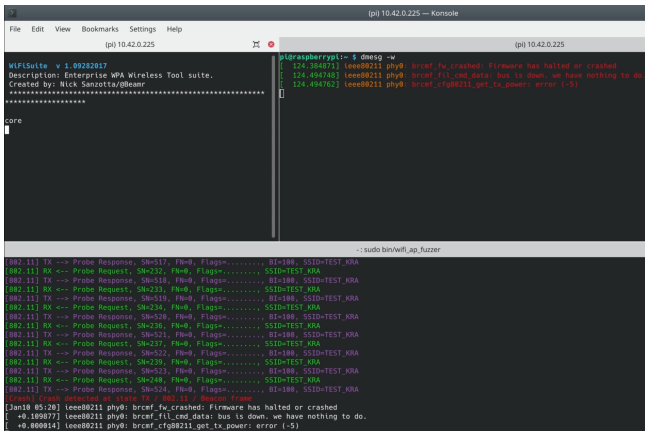

Figure 17: Raspberry Pi 3B Probe Response Deadlock (CVE-2022-26599)

Similarly to BLE Host, you can optionally run the Wi-Fi AP fuzzer in normal mode and replicate Table 7 as follows:

```
sudo bin/wifi_ap_fuzzer --mutation=true
--duplication=true --optimization=true --max-iterations=1\
    000
cp logs/wifi_ap modules/eval/custom
cd modules/eval
./eval.py custom
```

The customization of this experiment to target other devices as shown in Table 2 is discussed in A.7.5.

A.7 Experiment Customization

A.7.1 Experiment 1

To replicate all the results of Table 4 of the paper, you can launch the BT fuzzer with the following arguments:

```
sudo bin/bt_fuzzer --no-gui --target=<BDAddress> \
     --target-port=<Serialport> --duplication=true \
     --mutation=true --optimization=true
cp logs/Bluetooth modules/eval/custom
cd modules/eval
./eval.py custom
```

The arguments **–target** and **–target-port** corresponds to the columns *BDAddress* and *Monitor* of Table 1. Note that **–target-port** is only used if the target is using the *Serial* monitor type. For targets that use *SSH* or *ADB*, the file *configs/bt_config.json* need to be updated as follows:

- SSH:
  Update attribute *"MonitorType":1*
  Update attribute *"SSHUsername": "artifact"*
- ADB:
  Update attribute *"MonitorType":3*
  For Oppo Reno 5G, update attribute *"ADBDevice": "627ff0eb"*
  For OnePlus 5T, update attribute *"ADBDevice": "3ffd4d9a"*

After the changes above, you can start the fuzzer without argument *–target-port*.

Next, to replicate all the results of Table 5 of the paper with the correct number of Iterations and parameters (R_{sel} and D_T), the configuration file *configs/bt_config.json* can be changed as follows:
(*i*) "MaxIterations": 200
(*ii*) "DefaultDuplicationProbability": R_{sel}
(*iii*) "MaxDuplicationTime": D_T

A.7.2 Experiment 2

To generate your own BT captures to be used in state machine model generation (Experiment 2), you can disable the fuzzer components and increase the global timeout as follows:
(*i*) "enable_duplication": false
(*ii*) "enable_mutation": false
(*iii*) "enable_optimization": false
(*iv*) "GlobalTimeout": 9999 Then you can start the fuzzer without the argument *–mutation, –duplication, –optimization*. Example:

```
sudo bin/bt_fuzzer --no-gui --mutation=false
--duplication=false --optimization=false
```

Finally, after running the tool for a while, the reference wireshark captures are saved to *logs/Bluetooth/capture_bluetooth.pcapng*

A.7.3 Experiment 4

To run the tools against other BT devices from Table 1, use their respective *BDAddress* when starting each comparison script. For example, you need to manually modify script *experiment3_bss.sh:25* to use the correct *BDAddress* of the target device. For the other comparison scripts, you need to wait for the tools to scan the environment before selecting the correct *BDAddress* of the target BT device.

A.7.4 Experiment 5

In order to launch exploits to trigger the bugs reported in Table 2 of the paper (*Summary of unknown implementation bugs and other anomalies found*), we need the mapping between the Exploit Name and the vulnerability name as reported in Table 2 of the paper. Table 3 outlines the columns **Attack Name, Exploit Name**, CVE ID, Affected SoC, etc to capture this mapping.

An exploit can be launched from the fuzzer with the following arguments:

```
sudo bin/bt_fuzzer --no-gui --target=<BDAddress> \
     --target-port=<Monitor>
--exploit=<Exploit Name>
```

Note that the **<BDAddress>,<Monitor>** corresponds to Table 1 and **<Exploit Name>** corresponds to the last column of Table 3. Lastly, the argument *–target-port=<Monitor>* is optional and can be configured for ADB and SSH targets as discussed in Section A.7.1.

You can also list all the available exploits names that are stored in folder *modules/exploits/bluetooth/*.cpp* by running the following command:

```
sudo bin/bt_fuzzer --no-gui --list-exploits
```

Exploit creation and modification is out of the scope of this artifact, but a tutorial material is included in the documentation file *exploit_modules_tutorial.pdf* at the root folder of the artifact package.

A.7.5 Experiment 6

In order to evaluate the fuzzer against the BLE Targets of Table 2, you can launch the BTHost fuzzer as follows:

```
sudo bin/bthost_fuzzer --target=<Address> --target-port=<\
     Monitor> --duplication=true --mutation=true \
     --optimization=true
```

The parameter **Address** is informed by Table 2 and devices in which the column *Monitor* is *"N.A"* means that no monitor is applicable to such device. In this case, you can omit the argument *"–target-port"* before launching the fuzzer.

Similarly to the BTHost fuzzer, you can launch the Wi-Fi fuzzer as follows:

```
     sudo bin/wifi_ap_fuzzer --target-port=<Monitor> \
     --duplication=true --mutation=true --optimization=\
     true
```

Note that since the Wi-Fi fuzzer is a rogue AP which waits a connection from the target device, the column *Address* of Table 2 is not applicable.

Similar to the monitor configuration procedure of Section A.7.1, the argument **–target-port** is only used if the target is using the *Serial* monitor type. For targets that use *SSH* or *ADB*, the file *configs/wifi_ap_config.json* or *bthost_config.json* (depending on which fuzzer you are running) need to be updated as follows:

- SSH:
  Update attribute *"MonitorType":1*
  For Raspberry Pi, update attribute *"SSHUsername": "pi"*
  For Raspberry Pi, update attribute *"SSHHostAddress": "10.42.0.220"*
- ADB:
  Update attribute *"MonitorType":3*
  For OnePlus 5T, update attribute *"ADBDevice": "3ffd4d9a"*

After the changes above, you can start the fuzzer without argument *–target-port*.

Table 3: Summary of Exploits and Affected BT Devices

CVE ID	Attack Name	Affected Vendor(s)	Affected SoC(s) or Product(s)	Impact	Exploit Name
CVE-2021-28139	Feature Page Execution	Espressif Systems	ESP32 (SoC)	ACE / Deadlock	invalid_feature_page_execution
CVE-2021-28136	Duplicated IOCAP	Espressif Systems	ESP32 (SoC)	Crash (Reboot)	duplicated_iocap
CVE-2021-28135	Feature Res. Flooding	Espressif Systems	ESP32 (SoC)	Crash (Reboot)	feature_response_flooding
CVE-2021-28138	Invalid Public Key	Espressif Systems	ESP32 (SoC)	Crash (Reboot)	wrong_encapsulated_payload
CVE-2021-28137	Feature Req. Ping-Pong	Espressif Systems	ESP32 (SoC)	Crash (Reboot)	feature_req_ping_pong
CVE-2021-28155	Feature Res. Flooding	Harman International	JBL TUNE500BT (Product)	Crash (Shutdown)	feature_response_flooding
CVE-2021-31609	LMP Auto Rate Overflow	Silabs	WT32i (SoC)	Crash (Reboot)	lmp_auto_rate_overflow
CVE-2021-34147	Invalid Timing Accuracy	Infineon Technologies	CYW20735B1 (SoC)	Crash (Reboot)	invalid_timing_accuracy
CVE-2021-34146	AU Rand. Flooding	Infineon Technologies	CYW20735B1 (SoC)	Crash (Reboot)	au_rand_flooding
CVE-2021-34145	LMP Invalid Max Slot Type	Infineon Technologies	CYW20735B1 (SoC)	Crash (Reboot)	invalid_max_slot
CVE-2021-34148	LMP Max Slot Overflow	Infineon Technologies	CYW20735B1 (SoC)	Crash (Reboot)	lmp_max_slot_overflow
CVE-2021-34149	AU Rand. Flooding	Texas Instruments	CC2564C (SoC)	Deadlock	au_rand_flooding
CVE-2021-31610	AU Rand. Flooding	Bluetrum	BT889X / AB5XX / AB5301A (SoCs)	Crash (Reboot)	au_rand_flooding
CVE-2021-34150	LMP Length Overflow over DM1	Bluetrum	AB5301A (SoC)	Deadlock (Paging disabled)	lmp_overflow_dm1
CVE-2021-34143	AU Rand. Flooding	Zhuhai Jieli Technology	AC6366C (SoC)	Deadlock	au_rand_flooding
CVE-2021-34144	Truncated SCO Link Request	Zhuhai Jieli Technology	AC6366C (SoC)	Deadlock	truncated_sco_link_request
CVE-2021-31612	LMP Auto Rate Overflow	Zhuhai Jieli Technology	AC6905X (SoC)	Deadlock	lmp_auto_rate_overflow
CVE-2021-31613	Truncated LMP *accepted*	Zhuhai Jieli Technology	AC6905X / AC6925C (SoC)	Crash (Reboot)	truncated_lmp_accepted
CVE-2021-31611	Invalid Setup Complete	Zhuhai Jieli Technology	AC6905X / AC6925C (SoC)	Deadlock	invalid_setup_complete
CVE-2021-31787	Feature Res. Flooding	Actions Technology	ATS2815 / ATS2819 (SoC)	Crash (Shutdown)	feature_response_flooding
CVE-2021-31785	Repeated Host Connection	Actions Technology	ATS2815 / ATS2819 (SoC)	Deadlock	repeated_host_connection
CVE-2021-31786	Multiple Same Host Connection	Actions Technology	ATS2815 / ATS2819 (SoC)	Deadlock (Shutdown)	N.A (Specific BDAddress Configuration)
CVE-2021-33155	LMP Paging Scan Disable	Intel	Intel AX200 (SoC)	Deadlock (Paging disabled)	paging_scan_disable
CVE-2021-33139	Invalid Timing Accuracy	Intel	Intel AX200 (SoC)	Crash (FW Reboot)	invalid_timing_accuracy
CVE-2021-30348	Invalid Timing Accuracy	Qualcomm	Snapdragon 845 / 855 / Others (SoCs)	Crash (FW Reboot)	invalid_timing_accuracy
CVE-2021-35093	LMP Length Overflow over 2-DH1	Qualcomm	CSR 8811 / CSR 8510 (SoCs)	Deadlock / Crash	lmp_overflow_2dh1
Pending	LMP Invalid Transport	Beken	BK3266	Deadlock (Paging disabled)	lmp_invalid_transport
CVE-2019-9506	Knob (Extra - For testing only)	Many	Many	Entropy Reduction	knob

A.8 Notes

In case that the BT target ESP32 hangs the fuzzing process and does not seem to move forward, then you can reset ESP32 by running the following command:

```
cd $HOME/braktooth/wdissector/modules/eval
./flash_esp32.sh
```

Due to IP requirements with our Keysight partners, the main source code of the fuzzer is freely available only for academic research purposes upon request to https://src.braktooth.com. Students or Researchers with a valid university email, will receive an automated invitation to our Gitlab repository. Nevertheless, the source code of our ESP32 reverse engineering framework is available to public at https://github.com/Matheus-Garbelini/esp32_firmware_patching_framework.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.