DEMO: Using NexMon, the C-based WiFi firmware modification framework

Matthias Schulz
Secure Mobile Networking Lab
TU Darmstadt, Germany
mschulz@seemoo.de

Daniel Wegemer
Secure Mobile Networking Lab
TU Darmstadt, Germany
dwegemer@seemoo.de

Matthias Hollick
Secure Mobile Networking Lab
TU Darmstadt, Germany
mhollick@seemoo.de

ABSTRACT

FullMAC WiFi chips have the potential to realize modifications to WiFi implementations that exceed the limits of current standards or to realize the implementation of new standards, such as 802.11p, on off-the-shelve hardware. As a developer, one, however, needs access to the firmware source code to implement these modifications. In general, WiFi firmwares are closed source and do not allow any modifications. With our C-based programming framework, NexMon, we allow the extension of existing firmware of Broadcom's FullMAC WiFi chips. In this work, we demonstrate how to get started by running existing example projects and by creating a new project to transmit arbitrary frames with a Nexus 5 smartphone.

1. INTRODUCTION

WiFi chips are mainly offered in two variants: SoftMAC chips that outsource time uncritical tasks into the driver and FullMAC chips that implement the complete medium access control (MAC) layer in the WiFi chip and only exchange Ethernet frames with the driver. In this work, we focus on the second category. These chips are mainly used in smartphones. They pursue the goal to relieve the main processor from handling and processing every received frame. The firmware not only offers to exchange Ethernet and WiFi frame headers, it also supports automatic address resolution protocol (ARP) responses and transmission control protocol (TCP) offloading.

Even though, WiFi manufacturers may offer a larger range of capabilities to developers and open up their firmwares similarly to existing open source SoftMAC drivers, such as bcrmsmac, manufactures keep firmwares locked and do not even offer datasheets that fully describe their internal chip architectures. The latter missing information includes the internal memory layout with memory mapped peripherals such as direct memory access (DMA) controllers, debug registers and other chip control registers.

In previous works, such as monmob [1] and bcmon [2], developers patched parts of existing WiFi firmwares of Broadcom's BCM432x and BCM4330 chips to enable monitor mode and frame injection on iPhones and Android smartphones. Their works are currently not only used for WiFi penetration testing using mobile devices, but also in the research community to try out new MAC-layer communication protocols on smartphones. Even though, monmob and bcmon opened up access to the MAC-layer on smartphones, the patches them-selves are closed source and, hence, not easily extensible. For example, a new mesh implementation may rather focus on mesh frames than on processing all received frames of a WiFi receiver running in monitor mode. Using the latter requires to drop many received frames in the operating system which is less energy efficient than dropping them directly in the firmware.

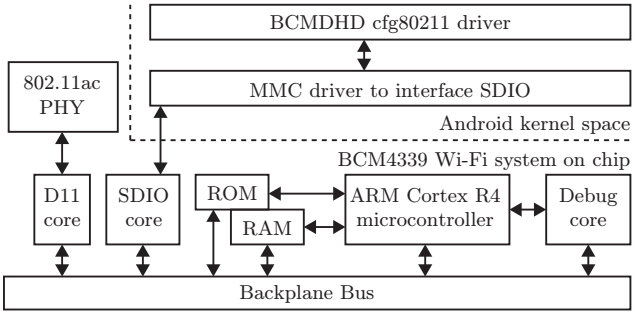

Figure 1: All Broadcom WiFi system-on-chips have a similar architecture. On SoftMAC chips, the D11 core is directly accessible by the driver, while on FullMAC chips an ARM processor arbitrates between driver and D11 core.

With NexMon [3] we offer a framework to modify the WiFi firmware of Nexus 5 smartphones with Broadcom BCM4339 chips (but are not limited to this platform). As illustrated in Fig. 1, all of Broadcom's WiFi chips have a similar architecture. It consists of an interface to the driver (here SDIO), a physical layer core and a D11 core which is a real-time capable programmable state machine. Modifications to the D11's firmware are illustrated in [4]. Compared to SoftMAC chips, FullMAC chips also include an ARM processor that runs a firmware similar to the bcrmsmac driver on Linux. It is used to process received frames from the D11 core and forward them as Ethernet frames to the driver, as well as to process frames from the driver and send them out using the D11 core. In this work, we demonstrate how to use the NexMon framework to modify a chip's firmware.

WiSec'16 , July 18-22, 2016, Darmstadt, Germany

ACM ISBN 978-1-4503-4270-4/16/07.

DOI: http://dx.doi.org/10.1145/2939918.2942419

In the following section, we first introduce the framework and then present some examples that are also available on our project website[1]. In the appendix, we describe how a conference participant can interact with our demonstrator.

2. PATCHING FRAMEWORK

The NexMon patching workflow is illustrated in Fig. 2. The patch code resides in the *patch.c* file. The compiler is instructed to create separate sections for each symbol, that means functions and global variables. The linker uses the *patch.ld* file to place the patch functions at defined locations. Symbols we intend to place by ourselves result in separate sections, other symbols are gathered in the *text* section. To call other functions existing in the firmware, the linker needs to know their locations to create correct branch instructions. To define those locations, we use the *wrapper.h* file, which contains function prototypes and addresses of the locations of those functions. From the header file, we create a *wrapper.c* file containing dummy function stubs and a *wrapper.ld* file to place the dummy functions using the linker. When linking the *patch.o* file to the *wrapper.o* file, the resulting *wrapper.elf* file contains the correct branch instructions as well as symbols from the wrapper and the patch files at the correct addresses. To only insert the sections of our patch into the resulting firmware, we need to extract each section from the elf-file into separate binary files. Then, we integrate those files into the original firmware binary using a Python script called patcher.py.

3. EXAMPLE PATCHES

On our project website, we offer multiple example projects that one can test on Nexus 5 smartphones. The *hello_world_example* project simply illustrates how to print on the chip's console and read the result in Android user space. The *monitor_mode_example* shows how to activate promiscuous mode and forward each received WiFi frame directly to the driver without further processing. To analyze firmware code in RAM and ROM, we offer the *debugger_example*. It sets hardware breakpoints to redirect program execution at a breakpoint into a handler function that can read and change register values. This can, for exmaple, be used to analyze function arguments of functions residing in ROM or to perform single-step debugging to figure out, where errors occur during execution.

[1]NexMon project: https://dev.seemoo.tu-darmstadt.de/bcm/bcm-public

4. ACKNOWLEDGMENTS

This work has been funded by the German Research Foundation (DFG) in the Collaborative Research Center (SFB) 1053 "MAKI – Multi-Mechanism-Adaptation for the Future Internet", by LOEWE CASED, LOEWE NICER, and by BMBF/HMWK CRISP.

5. REFERENCES

[1] A. Blanco and M. Eissler. One firmware to monitor 'em all., 2012.

[2] O. Ildis, Y. Ofir, and R. Feinstein. Wardriving from your pocket – Using wireshark to reverse engineer broadcom wifi chipsets, 2013.

[3] M. Schulz, D. Wegemer, and M. Hollick. NexMon: A Cookbook for Firmware Modifications on Smartphones to Enable Monitor Mode. *arXiv:1601.07077*, 2015.

[4] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, and F. Gringoli. Wireless MAC processors: Programming MAC protocols on commodity hardware. In *Proc. of the 31st Annual IEEE International Conference on Computer Communications (INFOCOM)*, 2012.

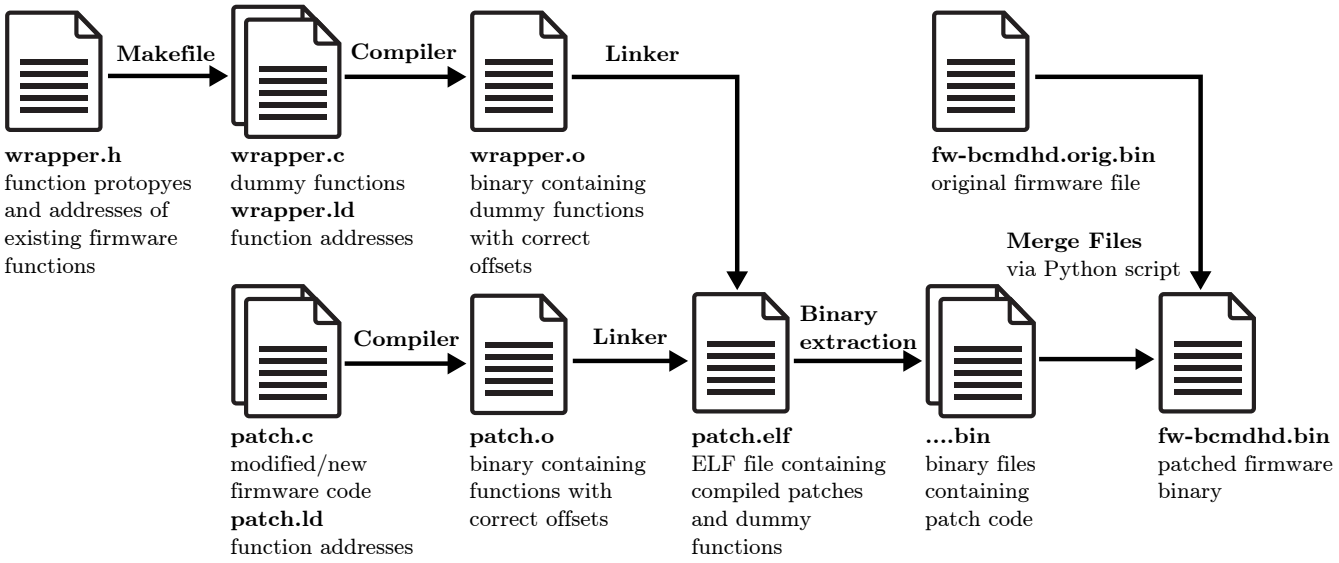

Figure 2: The NexMon firmware patching framework allows to write firmware patches in C or assember and to compile them into binary patches that can be linked to existing firmware functions.

APPENDIX

During the demonstration at the conference, we intend to show the participants how to get started with NexMon. Thereto, we bring a couple of Nexus 5 smartphones and laptops to program them. In addition, we intend to support participants who want to try out NexMon on their own smartphones. After creating a patched firmware file, it is combined with a driver and added to a *boot.img* file that also contains various binaries for penetration testing. A participant can use *fastboot* in bootloader mode to boot a Nexus 5 smartphone with the custom *boot.img*. This leaves the existing boot partitions in flash memory untouched and the phone can be rebooted into the original Kernel by a simple restart of the smartphone.

As a first test, the conference participants can take one of our example patches and execute them on the phone. The *monitor_mode_example* is a good starting point as it allows to run well known tools such as *tcpdump* or *airodump-ng* to observe and capture frames on the selected WiFi channel. The Nexus 5 supports single stream transmissions on channels with up to 80 MHz bandwidth in the 2.4 and 5 GHz WiFi bands following the 802.11ac standard and below.

To write a patch on their own, we instruct the conference participants to create their own "playground" project executing the following command in the *firmware_patching* directory:

```
make newproject NEWPROJECT=my_playground
```

This copies the *bcmdhd* driver as well as template *patch.c*, *patch.ld*, *Makefile*, and *patcher.py* files into a new *my_playground* directory. Then the conference participant has to select a function to hook so that our patch gets executed as soon as this function is called. We, therefore, propose the *wlc_radio_upd* function that sets up the physical layer core and activates the ability to transmit frames. We intend to overwrite the branch link instruction that calls *wlc_radio_upd* with a branch link instruction to our *wlc_radio_upd_hook* function that we insert into our *patch.c* file. The hook function should first call the original *wlc_radio_upd* function and then execute our own code before returning to the calling function. To achieve this, we write the *wlc_radio_upd_hook* function in assembler as it makes it easier to control how registers are used. We also need to save the link register before executing branch link instructions. Otherwise we cannot return to the calling function. The code looks as follows:

```
__attribute__((naked)) void wlc_radio_upd_hook(void) {
    asm("push {lr}\n"
        "bl wlc_radio_upd\n"
        "push {r0-r3}\n"
        "bl wlc_radio_upd_hook_in_c\n"
        "pop {r0-r3}\n"
        "pop {pc}\n");
}
```

This patch function should be placed in the free space starting at 0x180020 in the firmware. This is achieved by the following line in the *linker.ld* file:

```
.text.wlc_radio_upd_hook 0x180020: ↩
    { KEEP(patch.o (.text.wlc_radio_upd_hook)) }
```

This will only place the function in the object and elf-files. To extract the binary files, we need to set the *FUNCTIONS* variable in the Makefile to *wlc_radio_upd_hook*. To insert the binary file into the firmware, we need to insert the following line in the patcher.py file:

```
ExternalArmPatch(getSectionAddr( ↩
    ".text.wlc_radio_upd_hook"), ↩
    "wlc_radio_upd_hook.bin"),
```

As stated above, we intend to call our patch instead of the original function by replacing the branch link instruction at address 0x195B48 in the *WLC_UP* ioctl handler. Thereto, we insert the following line in the *patcher.py* file:

```
BLPatch(0x195B48, getSectionAddr( ↩
    ".text.wlc_radio_upd_hook")),
```

At this point, we can almost test, if the patch is working, but we are still missing the *wlc_radio_upd_hook_in_c* function called by our hook. As a start, we can simply insert a *printf* instruction as follows into the *patch.c* file:

```
void wlc_radio_upd_hook_in_c(void) {
    printf("hello world\n");
}
```

As we do not need to place this function at a specific address, the linker places it in the common *.text* section. To insert this section into the firmware, we need to uncomment the corresponding line in the *patcher.py* file. To test the new firmware, we run the following commands in the root directory of the NexMon project:

```
make boot
make reloadfirmware FWPATCH=my_playground
```

This reboots the smartphone with the built *boot.img* and copies the patched firmware as well as the corresponding driver module to the SD card. Here, it gets loaded as a kernel module. To load the firmware into the WiFi chip, we need to setup the *wlan0* interface and can then print the console of the chip using *dhdutil*:

```
adb shell "su -c 'ifconfig wlan0 up && ↩
    dhdutil -i wlan0 consoledump'"
```

In the output, one should see the hello world message. If it works, we continue to write the following code to create a new sk_buff, reserve space for additional headers, copy a beacon frame into the data variable and create a new station control block (SCB), which is required to transmit a frame through *wlc_sendctl*.

```
char pkt[] = {
    0x80, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff,
    0xff, 0xff, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc,
    0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0x10, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x64, 0x00, 0x21, 0x05, 0x00, 0x06,
    'N', 'E', 'X', 'M', 'O', 'N' // SSID
};

void wlc_radio_upd_hook_in_c(void) {
    sk_buff *p; void *scb;
    struct wlc_info *wlc = WLC_INFO_ADDR;
    void *bsscfg = wlc_bsscfg_find_by_wlcif(wlc, 0);
    p = pkt_buf_get_skb(wlc->osh, sizeof(pkt) + 202);
    p->data += 202; p->len -= 202;
    memcpy(p->data, pkt, sizeof(pkt));
    scb = __wlc_scb_lookup(wlc, bsscfg, pkt, 0);
    wlc_scb_set_bsscfg(scb, bsscfg);
    wlc_sendctl(wlc, p, wlc->active_queue, scb,
        1, 0, 0);
}
```

Running this code sends out a beacon frame announcing the service set identifier (SSID) NEXMON. One can, for example, use *tcpdump* to receive this frame on a nearby device listening on WiFi channel 1 and filtering for the host address cc:cc:cc:cc:cc:cc.