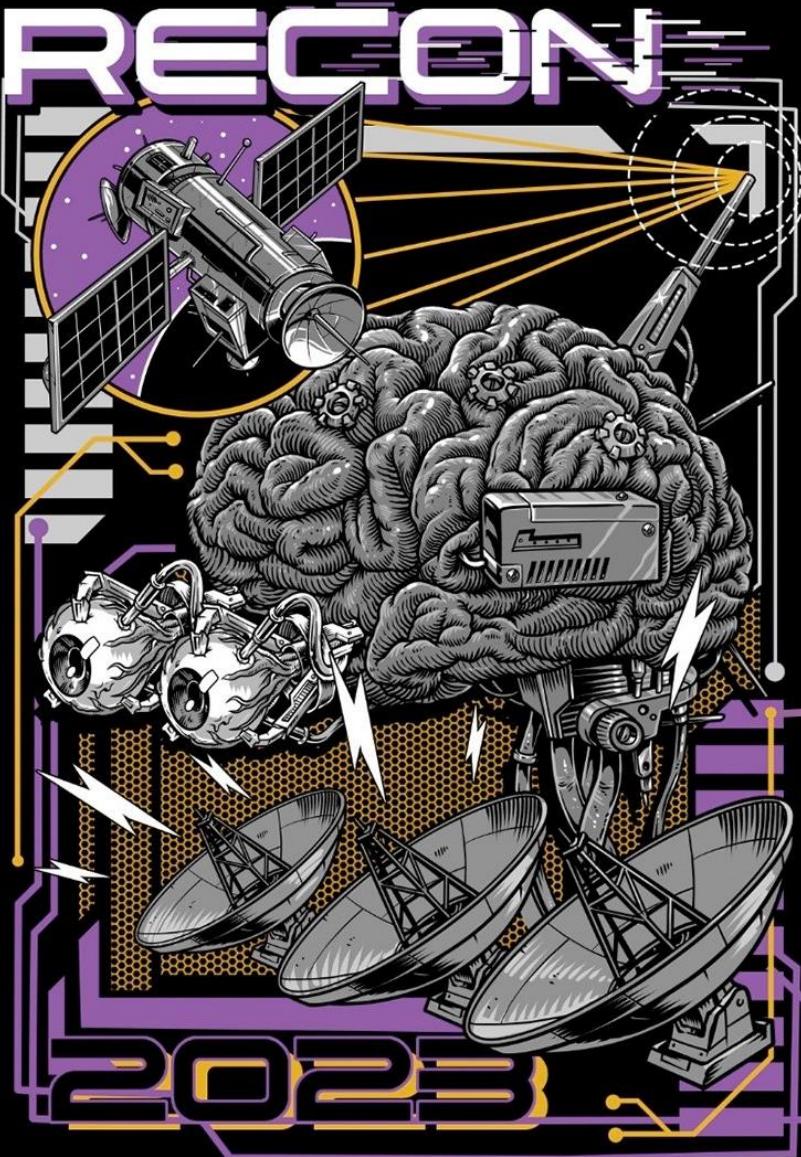


Enabling Security Research on Qualcomm Wifi Chips

RECON Montreal 2023 –
Daniel Wegemer

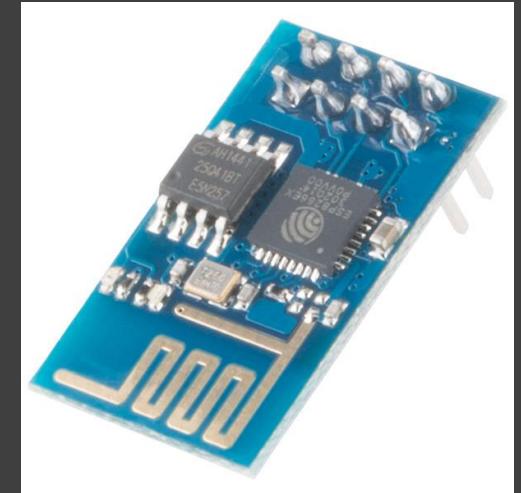


(Qualcomm logo according to Bing AI)



Motivation

- Wifi chips contain powerful processors
- These processors allow **general purpose computing**
 - Proprietary binaries prohibit running your own code
- Modifying the existing firmware can:
 - Enable additional functionality
 - Enable security research (dynamic analysis)

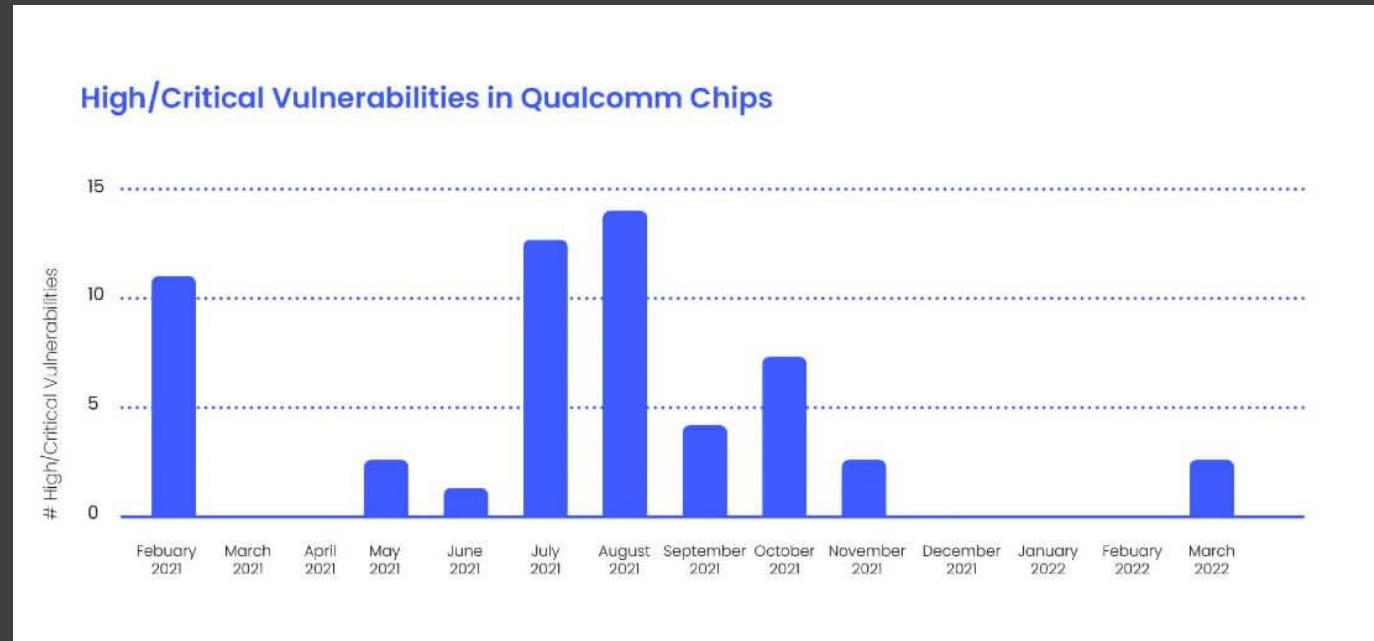


https://commons.wikimedia.org/wiki/File:WiFi_Module_-_ESP8266_%2816730689880%29.jpg

Motivation

Vendors of Wifi chips:

- **Qualcomm**
- Broadcom
- Intel
- Mediatek
- Texas Instruments
- NXP (former Marvell)

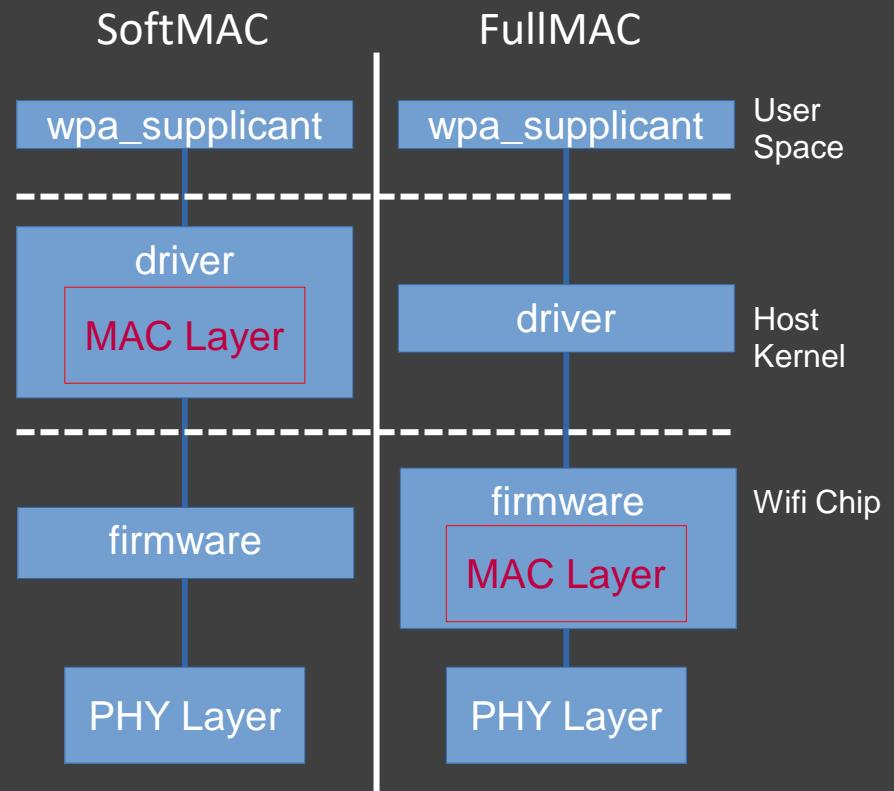


<https://aireye.tech/2022/03/29/trends-in-wi-fi-vulnerabilities-this-time-its-qualcomm/>

Qualcomm Wifi chips had many high or critical rated vulnerabilities in the past. **How many other vulnerabilities are there?**

Motivation

- Two types of Wifi chips: FullMAC and SoftMAC
- FullMAC implements MAC layer on-chip, often used for IoT and portable device
 - Much bigger firmware
 - Making changes to the driver does not change the behavior of (FullMAC) Wifi chips
 - We need to change the Wifi chip firmware directly



Previous Work

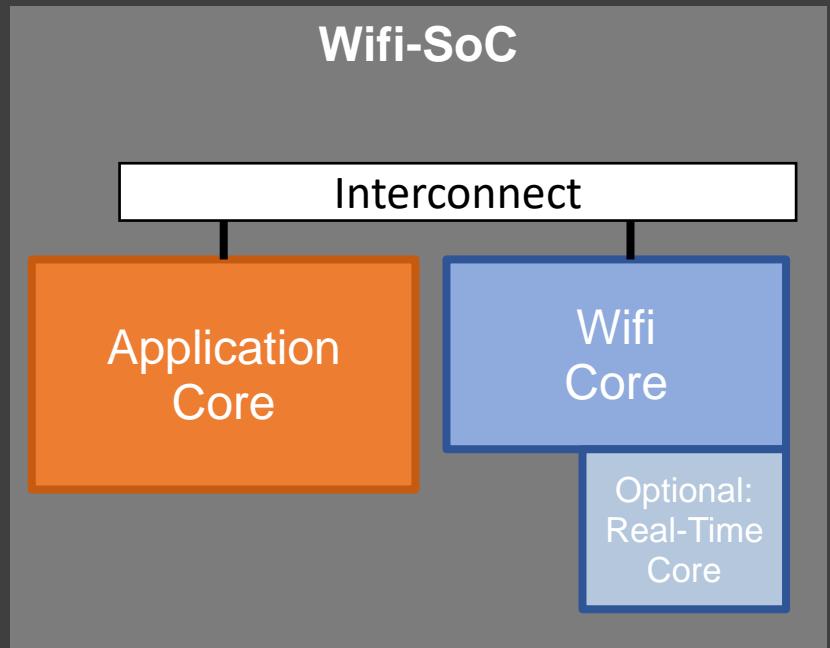
- Broadcom Wifi Chips: **Nexmon** Framework
 - Allows Wifi firmware patching on many Broadcom chips
 - Deep modifications on Wifi subsystem possible
 - See: <https://nexmon.org>
- **Intel** chips at Blackhat 2022:
“Ghost in the Wireless, iwlwifi Edition”
- Qualcomm **Hexagon** based chips at DEFCON 27 and Blackhat 2019:
“Exploiting Qualcomm WLAN and Modem Over the Air”

→ This is the first work on **Xtensa based** Qualcomm Wifi firmware

Background

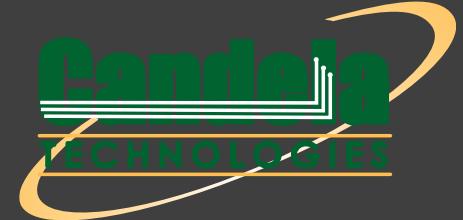
Wifi/IoT Chipset Overview

- SoC bundles multiple processors into a single package
- Wifi enabled devices often contain an **application processor** and a **processor** responsible for **Wifi**
- Timing critical Wifi functionality is sometimes handled by an additional processor



Three types of driver and firmware

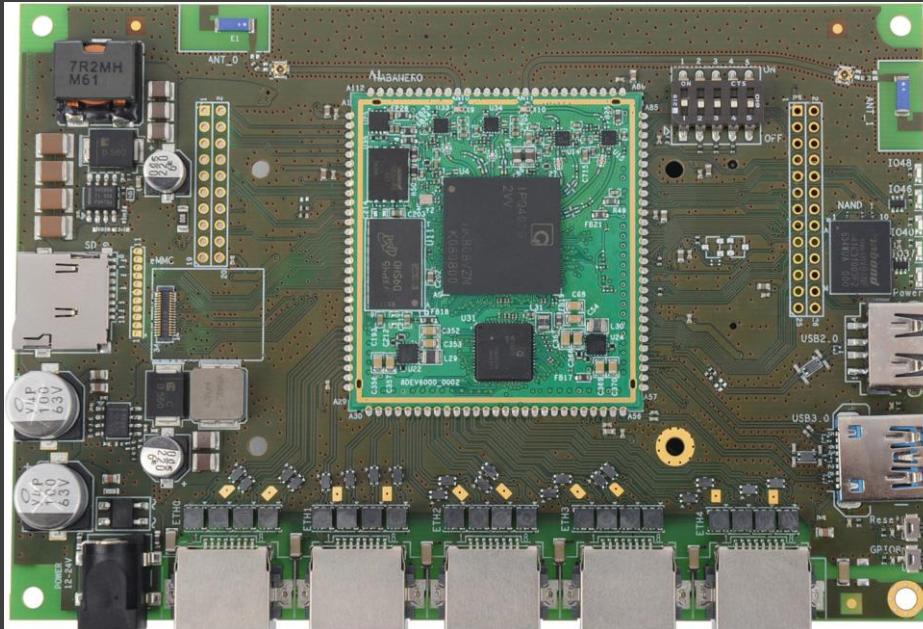
- **ath10k** by Qualcomm
- **ath10k-ct** by Candelatech
- **qcacld** by Qualcomm
→ Used for factory processes



<https://www.candelatech.com/>

ath10k-ct driver can also run QCA firmware

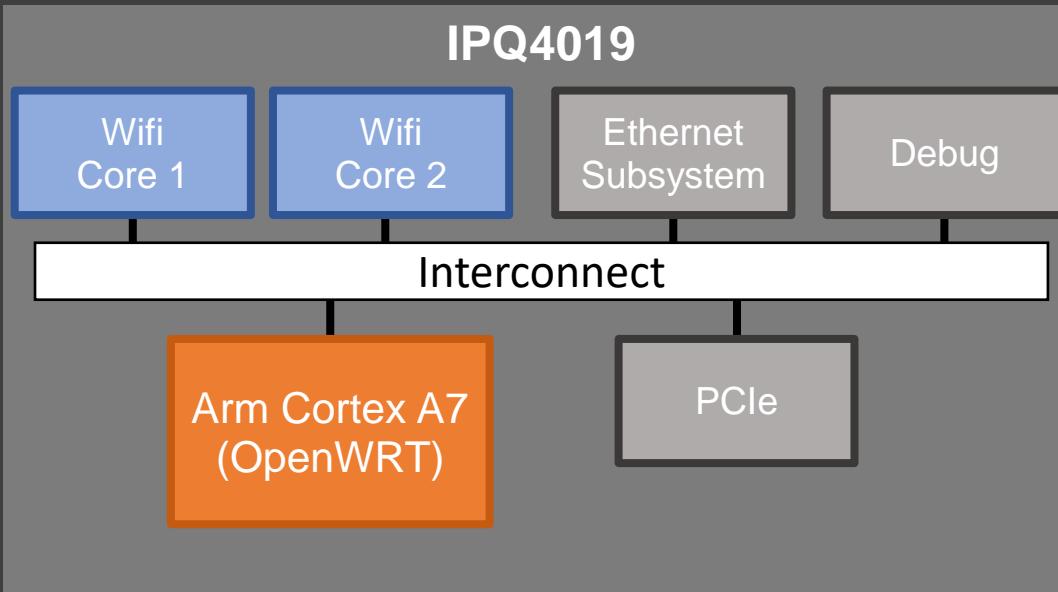
IPQ4019 ("Habanero" by 8devices)



<https://lian-mueller.com/media/catalog/product/cache/1/image/9df78eab33525d08d6e5fb8d27136e95/h/a/habanero-dvk-top.jpg>

IPQ4019

- Used mainly in Wifi home routers (e.g. AVM FritzBox)
- Application Core: ARM Cortex A7 → Runs OpenWRT 19.07 on Kernel 4.14
- Wifi Subsystem: Cadence/Tensilica Xtensa
 - 2.4 and 5 GHz
 - Uses PCIe to communicate



Firmware

Firmware: Overview

- Xtensa, little endian
- ROM + RAM
- RAM part of a file in the OpenWRT filesystem
 - Contains multiple “segments” (aka “SGMT”)
 - lz77 compressed
- ROM can be patched
- Codeswap: mechanism to put code of Wifi subsystem in the hosts memory
- No security enabled by default
 - No secure boot
 - No stack canneries
 - No address randomization



<https://www.cadence.com>

Firmware: Debugging

Debugfs: /sys/kernel/debug/ieee80211/phy0/ath10k/...

→ .../mem_value can be used for memory access r+w
(only works after core+pci kernel modules are loaded)

→ .../debug_mask, can be used to increase verbosity.
Also possible via module parameter “debug=mask=0xYYY” in ath10k_core.ko

ath10k: Add the target register access and memory dump debugfs interface

Message ID 1416656922-6645-1-git-send-email-yanbol@qti.qualcomm.com ([mailing list archive](#))
State Superseded, archived
Headers [show](#)

Commit Message

Yanbo Li

The debugfs interface reg_addr®_val used to read and write the target register.
The interface mmem_val used to dump the target memory and also can be used to assign value to target memory

The basic usage explain as below:

Register read/write:
reg_addr (the register address) (read&write)
reg_value (the register value, output is ASCII) (read&write)

Firmware: Interfaces

Interfaces between driver and Wifi subsystem

- **BMI** (Bootloader Messaging Interface)
 - Communicaton between host and Wifi subsystem during bootup
 - Implemented in ROM
- **WMI** (Wireless Module Interface)
 - Communicaton between host and Wifi subsystem after bootup
 - Example commands: wifi scanning, channel configuration etc...

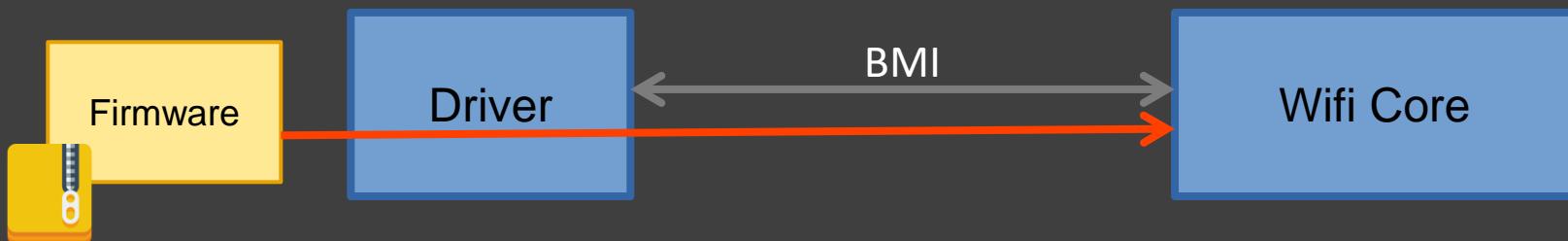
Firmware: Loading

Two loading methods possible:

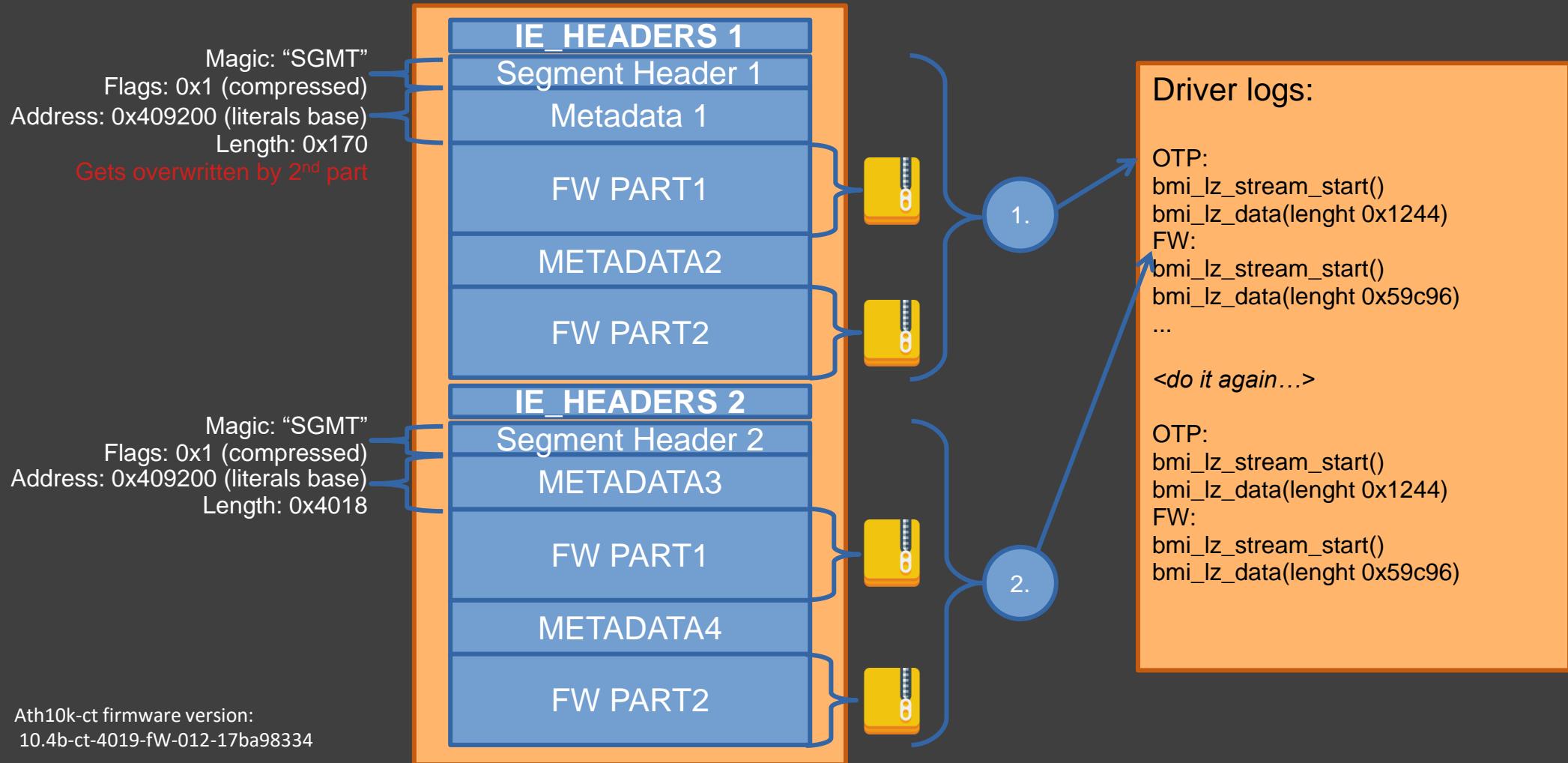
- BMI (Bootloader Messaging Interface)
- Copy Engine: `ath10k_hw_diag_fast_download()` in `ath10k`, different in `ath10k-ct`

In case of **compressed** firmware:

- BMI method needs to be used
- Decompression done in ROM of Wifi Core

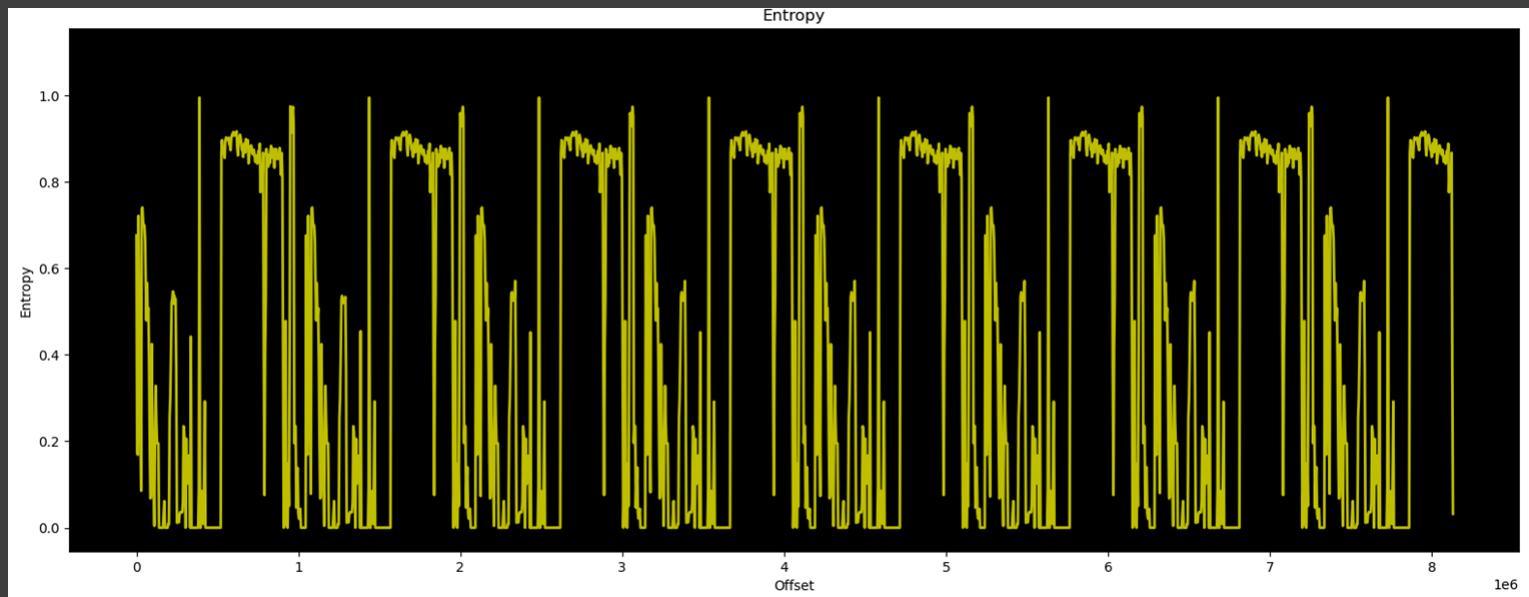


Firmware: File Format



Firmware: Memory Layout

Memory contents are repeating:



Ath10k-ct firmware version: 10.4b-ct-4019-fW-012-17ba98334

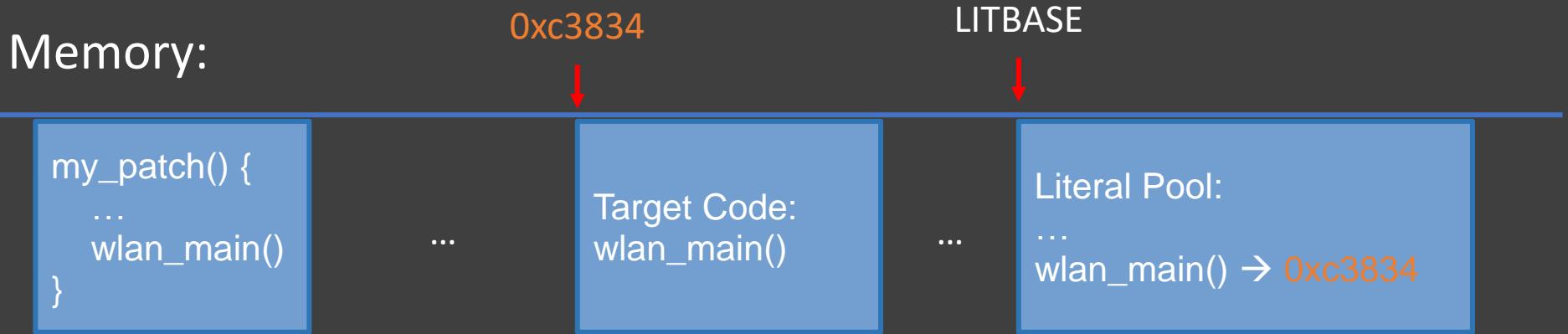
Xtensa Architecture

Xtensa in QCA Wifi chips

- **Literal Pools** are used for “L32R” instructions
 - Loads are **independent** of PC
 - Instead the offset is calculated from a fixed “LITBASE”
- **Call8** is used to move a register window by 8
 - e.g. a10 of caller will be a2 of callee

Register	Usage
a0	Return address
a1	Stack Pointer
a2 - a7	Incoming arguments

Xtensa Literal Pools



“wlan_main()” will use a “L32R” instruction to get the target address:

LITBASE

+ offset within Literal Pool (part of L32R)

= Target Address

This calculation is done in every “**I32r**” instruction!

Xtensa Literal Pools

- **LITBASE** is set at the beginning of FW execution:
 - For IPQ4019 Literal pool start is at 0x408001
 - Code:
l32r a2, lib4_start + 0x40001
wsr a2, LITBASE
- Existing Wifi firmware code does expect the LITBASE to be set as shown above

Xtensa Litbase in Disassemblers



IDA 7.7 adds support of Xtensa, but ignores LITBASE



Ghidra supports Xtensa using this plugin (<https://github.com/yath/ghidra-xtensa>) but ignores LITBASE



Radare2 ignores LITBASE



Binary Ninja using this plugin (<https://github.com/zackorndorff/binja-xtensa>) ignores LITBASE too

Xtensa: Disassembler Plugin for Binary Ninja

- Based on <https://github.com/zackorndorff/binja-xtensa>

- Patch:

```
diff --git a/binja_xtensa/instruction.py b/binja_xtensa/instruction.py
index 7243f07..e7f1700 100644
--- a/binja_xtensa/instruction.py
+++ b/binja_xtensa/instruction.py
@@ -34,6 +34,8 @@ Link to the Xtensa docs/manual I was referencing:
 """
from enum import Enum

+LITBASE = 0x408000 + 0x40001
+
# https://stackoverflow.com/a/32031543
def sign_extend(value, bits):
@@ -233,7 +235,8 @@ class Instruction:

    def offset_I32r(self, addr):
        enc = sign_extend(self.imm16 | 0xFFFF0000, 32) << 2
-       return (enc + addr + 3) & 0xFFFFFFFF
+       return (LITBASE & 0xFFFFF000) + enc
```

Xtensa: Disassembler Plugin for Ghidra

- Based on: <https://github.com/Ebiroll/ghidra-xtensa>

- Patch:

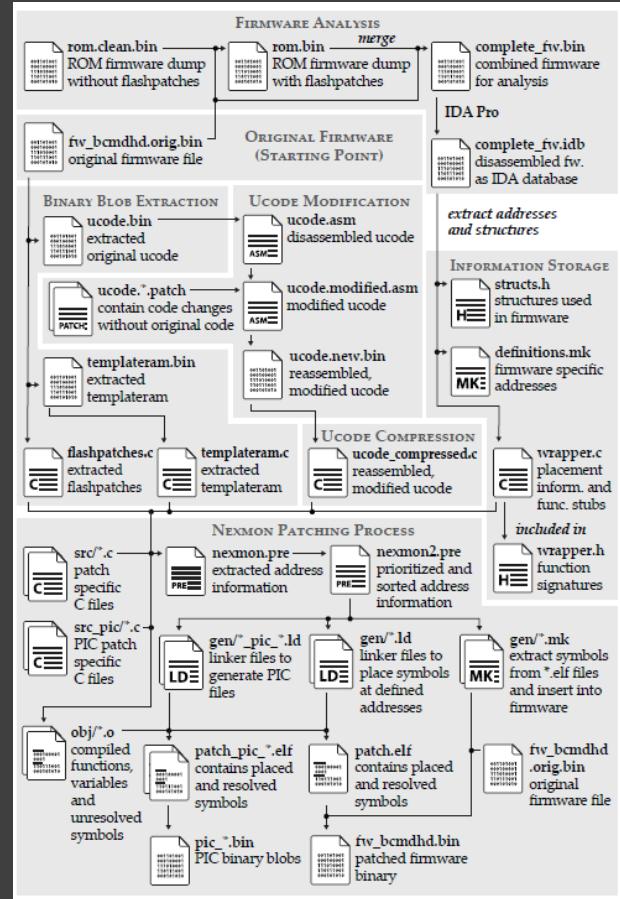
```
diff --git a/data/languages/xtensa.sinc b/data/languages/xtensa.sinc
index 80ac9bf..e8ef5c8 100644
--- a/data/languages/xtensa.sinc
+++ b/data/languages/xtensa.sinc
@@ -236,7 +236,7 @@ srel_6.23_sb2: rel is s8_6.23 [
 ] { export *:4 rel; }

srel_8.23_oex_sb2: rel is u16_8.23 [
-  rel = ((inst_start + 3) & ~3) + ((u16_8.23 | 0xffff0000) << 2);
+  rel = (0x448001 & 0xFFFF0000) + ((u16_8.23 | 0xffff0000) << 2);
] { export *:4 rel; }
```

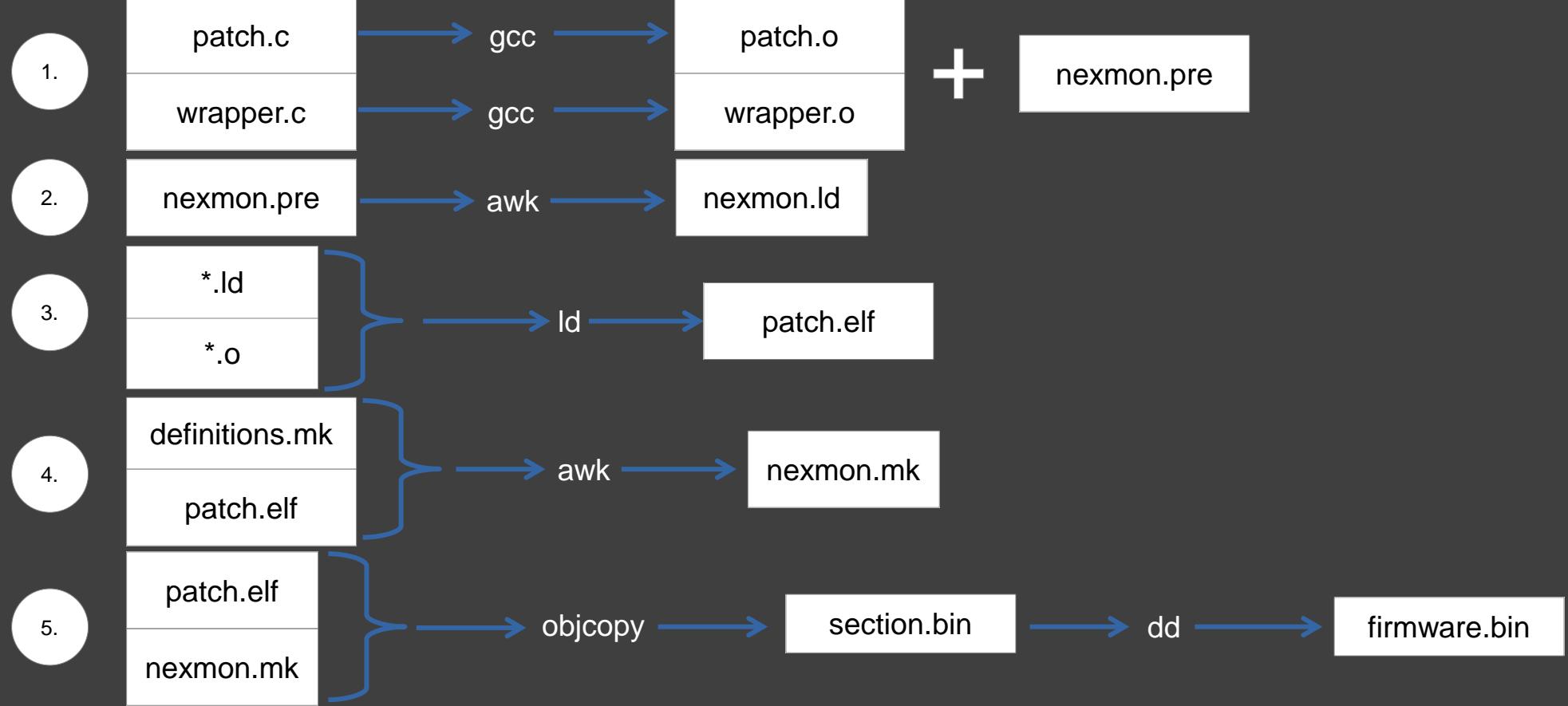
Firmware patching using Nexmon

Nexmon: Introduction

- Extract ROM, “Flashpatches”, RAM, UCODE for Broadcom Wifi chips
- Write patches in C, call existing firmware functions
- Compile and link firmware code
- Create firmware file



Nexmon: Introduction



Nexmon: Adaption

Necessary **changes** Qualcomm firmware:

1. Decompress segments from firmware-5.bin
2. Support multiple binaries (one for each segment)
3. Support “LITBASE”

Overview (steps in Makefile):

- Compiles + Links patches
- Copy patches into binary 0x980000 of 2nd (decompressed) segment
- Compress segment parts, create 2nd segment
- Adds padding bytes to 2nd segment
- Creates complete binary: firmware-5.bin

Nexmon: 2. Handle multiple binaries

- GCC Plugin is used to create “nexmon.pre” file
- This file is also used as input to dd & linker
- Extended to include a file name:

“attribute” in source code

```
__attribute__((at(0x409228, "", CHIP_VER_QCA, FW_VER_QCA, "segment2_00409200_mod.bin")))
```



nexmon.pre

0x00409228	PATCH	obj/patch.o	my_patch	segment2_00409200_mod.bin
------------	-------	-------------	----------	---------------------------

0x000c3834	DUMMY	obj/wrapper.o	wlan_main	segment2_00980000_mod.bin
------------	-------	---------------	-----------	---------------------------

Patching Firmware – 1st attempt

- Patch goal: write 0x1234 to an address, jump back to original code
- Use xtensa-esp32-elf-gcc to compile + link LE binary
- Load into IPQ4019 chip
- Use DebugFS to read memory after patch has run to check if 0x1234 was written successfully

→ Does not work! Why?

```
1 void my_patch(void *a1, void *a2, void *a3, void *a4) {  
2     int *p = (int *) 0x410a88;  
3     *p = 0x1234;  
4     wlan_main(a1, a2, a3, a4);  
5 }
```

Patching Firmware – 1st attempt

- L32R uses offset in LITBASE to calculate target address
- LITBASE is set at the very beginning of the FW execution in ROM
→ Existing code in FW relies on this
- There is no parameter to tell our compiler/linker where an already existing LITERAL POOL is!

```
entry      a1, 32
mov.n     a10, a2
mov.n     a11, a3
mov.n     a12, a4
mov.n     a13, a5
movi.n    a14, 0
132r      a8, fffc0014 <my_patch+0xffffc0014>
132r      a9, fffc0018 <my_patch+0xffffc0018>
s32i.n    a9, a8, 0
132r      a8, fffc001c <my_patch+0xffffc001c>
callx8   a8
retw.n
```

Patching Firmware – 1st attempt

Hack: Avoid L32R instructions

- Use immediate values only
- No references!
- Needs handcrafted Assembly



```
1 void my_patch(void *a1, void *a2, void *a3, void *a4) {
2     asm (
3         "mov.n    a10, a2\\n"
4         "mov.n    a11, a3\\n"
5         "mov.n    a12, a4\\n"
6         "mov.n    a13, a5\\n"
7
8         //a6 = 0x1234
9         "movi a6, 0x12\\n"
10        "slli a6, a6, 8\\n"
11        "movi a7, 0x34\\n"
12        "add a6, a6, a7\\n"
13
14        //a7 = 0x410a88
15        "movi a7, 0x41\\n"
16        "slli a7, a7, 16\\n"
17        "movi a8, 0x0a\\n"
18        "slli a8, a8, 8\\n"
19        "add a7, a7, a8\\n"
20        "movi a8, 0x88\\n"
21        "add a7, a7, a8\\n"
22        //store 0x1234 at 0x410a88
23        "s32i.n a6, a7, 0\\n"
24
25        //a6 = 0xc3834
26        "movi a6, 0xc\\n"
27        "slli a6, a6, 16\\n"
28        "movi a7, 0x38\\n"
29        "slli a7, a7, 8\\n"
30        "add a6, a6, a7\\n"
31        "movi a7, 0x34\\n"
32        "add a6, a6, a7\\n"
33
34        "callx8 a6\\n"
35    );
36 }
```

Nexmon: 3. Handle LITBASE

Two possible solutions:

1. Tell Linker where **existing Literal Base** is and how full it is
2. Use our **own LITBASE value**

→ We are going with option #2!

Necessary steps:

- Set LITBASE to 0x0 at function **entry**.
→ This can be done extending Nexmons GCC plugin
- Set LITBASE to original value at function **exit**.
→ This needs to be done in Binutils assembler because it expands/relaxes calls to a load+call

Nexmon: 3. Handle LITBASE

Patching Binutils Assembler “as”

- Needs to be done because of **“Instruction Relaxation”** in the assembler
- Relaxation adds a l32r before each call instruction
- The LITBASE used in this l32r needs to point to 0x0.
- Only **after this** we can set the LITBASE back to its original value

```
entry      a1, 32
mov.n     a10, a2
mov.n     a11, a3
mov.n     a12, a4
mov.n     a13, a5
movi.n    a14, 0
l32r      a8, fffc0014 <my_patch+0xffffc0014>
l32r      a9, fffc0018 <my_patch+0xffffc0018>
s32i.n   a9, a8, 0
l32r      a8, fffc001c <my_patch+0xffffc001c>
callx8   a8
retw.n
```

→ We can not patch this in GCC, its needs to be patched in the assembler!

Nexmon: 3. Handle LITBASE

The assembler uses “string patterns” to “relax” instructions:

```
static string_pattern_pair widen_spec_list[] =  
{  
    //...  
  
    /* Expanding calls with literals. */  
    {"call10 %label,%ar0 ? IsaUseL32R",      "LITERAL %label; 132r a0,%LITERAL; callx0 a0,%ar0"},  
    {"call14 %label,%ar4 ? IsaUseL32R",      "LITERAL %label; 132r a4,%LITERAL; callx4 a4,%ar4"},  
    {"call18 %label,%ar8 ? IsaUseL32R",      "LITERAL %label; 132r a8,%LITERAL; callx8 a8,%ar8"},  
    {"call12 %label,%ar12 ? IsaUseL32R",     "LITERAL %label; 132r a12,%LITERAL; callx12 a12,%ar12"},  
};
```

Nexmon: 3. Handle LITBASE

Patching Binutils Assembler “as”

```
static bool
xg_build_to_stack (IStack *istack, TInsn *insn, BuildInstr *bi)
{
    ...
    for ( ; bi != NULL; bi = bi->next)
    {
        /* QCAMON: IF CURRENT OPCODE is callx8 & PREV OPCODE is I32r */
        + if(bi->opcode == 0x3a && prev_bi->opcode == 0x86) {
        +     TInsn *new_insn = (TInsn *) malloc(sizeof(TInsn));
        +     tinsn_init (new_insn);
        +     build_wsr_litbase_insn(new_insn);
        +     new_insn->debug_line = insn->debug_line;
        +     new_insn->loc_directive_seen = insn->loc_directive_seen;
        +     istack_push(istack, new_insn);
        + }
        +
        TInsn *next_insn = istack_push_space (istack);

        ...
    return true;
}
```

Patching Firmware – 2nd attempt

- Patch goal: write 0x1234 to an address, jump back to original code
- C-Code can now be used!
- We can use GCC plugin + patched “as” to compile our code

→ Much simpler code

```
1 void my_patch(void *a1, void *a2, void *a3, void *a4) {  
2     int *p = (int *) 0x410a88;  
3     *p = 0x1234;  
4     wlan_main(a1, a2, a3, a4);  
5 }
```



Patching Firmware – 2nd attempt

Assembly compiled with our GCC plugin and the patched binutils assembler

```
1  entry          a1, 32
2  mov.n          a10, a2
3  mov.n          a11, a3
4  mov.n          a12, a4
5  mov.n          a13, a5
6  rsr.litbase   a15
7  movi.n         a14, 0
8  wsr.litbase   a14
9  l32r           a8, fffc0014 <my_patch+0xffffc0014>
10 l32r           a9, fffc0018 <my_patch+0xffffc0018>
11 s32i.n         a9, a8, 0
12 l32r           a8, fffc001c <my_patch+0xffffc001c>
13 wsr.litbase   a15
14 callx8         a8
15 retw.n
```

Open Problems

- Binutils patch needs implementation based on using Stack (avoid using a register)
- Support for disassemblers is lacking
- Missing text console in firmware for easier debugging
→ We can implement this ourselves now!

(lame) Demo Time!

Summary & Future Work

- Modified Nexmon framework can be found here: <https://qcamon.org>, including:
 - Demo patch
 - patches for Ghidra and Binary Ninja
 - GCC to complie LE Xtensa
 - Patched Binutils
- First POC patch code shows **feasibility of firmware modifications**
- **Further improvements** will help to make firmware modifications easier and enable security research
- Use “Production Software” (**QDART**) by Qualcomm to explore hidden FW functionality
- Explore “**Codeswap**” feature of FW

Thanks

- Martin Korth (aka ProblemKaputt)
for his awesome GBA reverse engineering
- rqou (aka ArcaneNibble)
for “ath10k_unzl.py” script

Q&A

Visit <https://qcamon.org> !

Mail: daniel@wegemer.com