

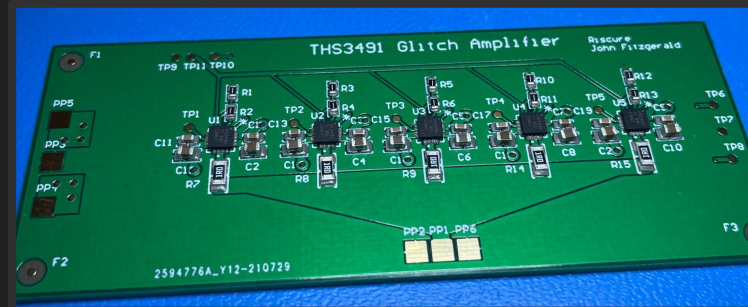
# Durablowl SH3002-RH

John Fitzgerald

# Introduction

- John Fitzgerald, Massachusetts
- Hardware hacker
- Software hacker
- Reverse engineer
- Circuit designer
- Sails boats, builds mini boats, and fixes old motors, among many other hobbies

<https://johnfitz.me/>



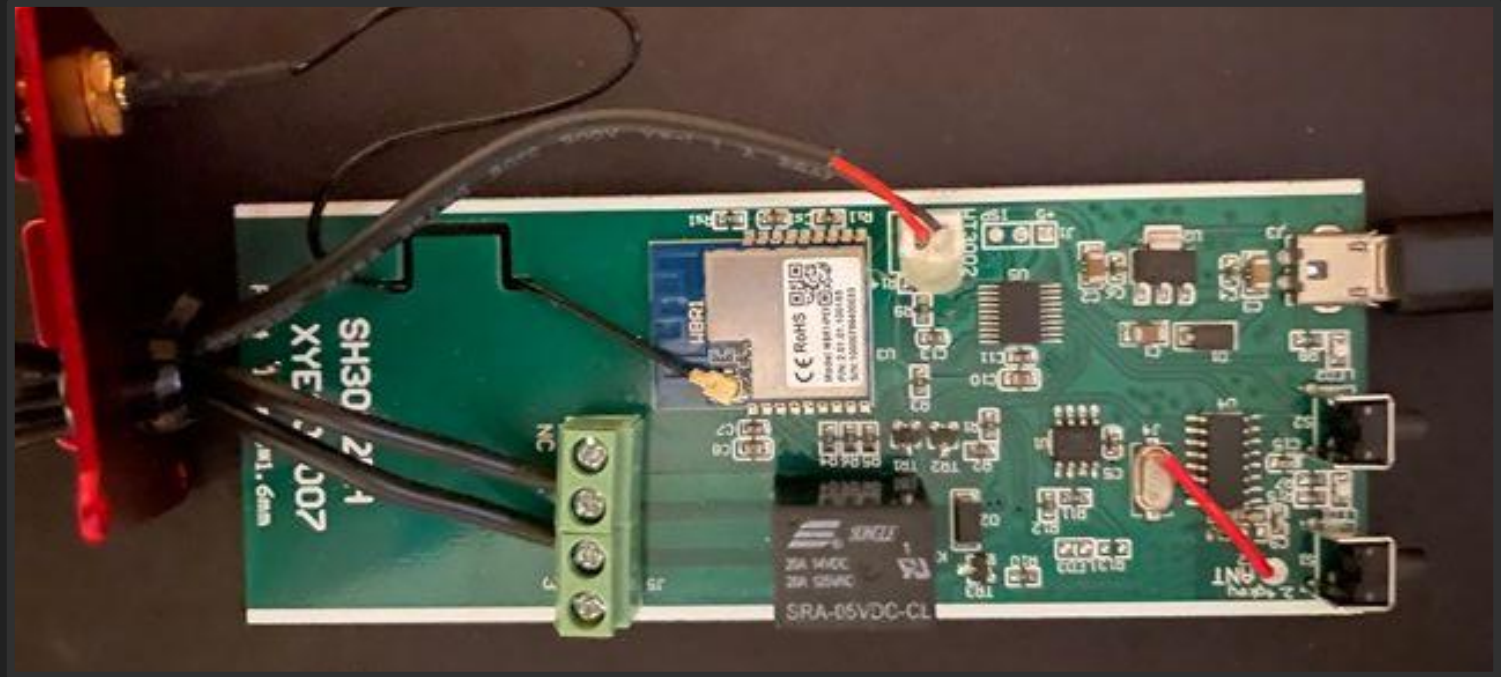
# SH3002-RH basic design

- Simple device designed to toggle a fireplace blower fan
- It uses a relay to accomplish this task
- Communicates over WiFi/BLE and a remote
- Use your phone with the application to heat your home
  - WAN or LAN

# Device setup

- Download Smart Life App
  - Register account and sign up
  - App is pretty basic, reverse engineering the android app yielded some interesting information
- Three ways to add the device
  - The device wants to connect to WiFi/cloud, so you need to give it credentials to your AP
  - Auto scan mode - uses Bluetooth? to detect device from app then gives it credentials
  - Manual AP - connect to the AP setup by the tuya module and send it credentials that way
  - Add manually Bluetooth - find the bluetooth device and then give it credentials to your AP

# Board overview and layout



# Research on WiFi/Bluetooth

I first sniffed packets by ARP poisoning the device. A few things were discovered...

- LAN packets for on/off have same encrypted data body, but different header and some form of a hash
- TLS communicate with the tuya backend is made with a pre-shared key (weak)

As for Bluetooth, I did not look much into this. I quickly replayed some things over Ubertooth but no luck. Did not look further as I wanted to do other things.

I looked at the Android App for a bit and discovered some secret values used for generating an HMAC key used in signatures of some data packets? (the LAN packets? cloud backend?)

# What about the remote?

I theorize the security for the remote control is sub-optimal. I did not end up taking my LimeSDR to it, but it would be an interesting demonstration. If it does implement some rolling code or other basic checksum/crypto, it may be fun to try to crack.



# The plan

I can probably do something over WiFi/Bluetooth/RF, but I wanted to demonstrate some some hardware vectors. It seemed the App over LAN sent packets to the device with a poor HMAC based replay protection/auth mechanism. The key for data encryption was not rolling either and so the packet body at least could be replayed. But I did not look into this much more...

After I undermine the hardware "security" and dump the firmware, I can find bugs or look into software side of things more later (i.e. .... App for iOS/Android or the firmware for the Tuya app, or other MCU firmwares).



# Tuya module

WBR1 is a low-power embedded Wi-Fi and Bluetooth module that Tuya has developed. It consists of a highly integrated RF chip (RTL8720CF) with an embedded Wi-Fi network protocol stack and robust library functions. WBR1 also contains a low-power KM4 multipoint control unit (MCU), WLAN MAC, 1T1R WLAN, 256 KB static random-access memory (SRAM), and 2 MB flash memory, and has extensive peripherals.

WBR1 is an RTOS platform that integrates all function libraries of the Wi-Fi MAC and TCP/IP protocols. You can develop embedded Wi-Fi products as required.

Figure 1-1 shows the WBR1 architecture.

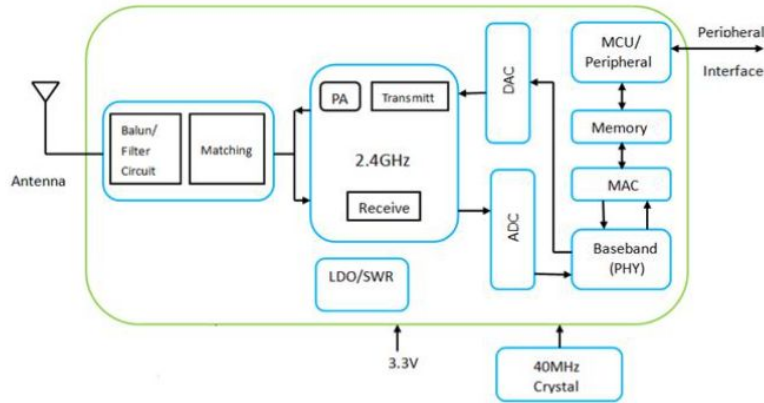


Figure 1-1 WBR1 architecture



## 14. Security Engine

### 14.1. Application scenario

The Ameba-Z II series security engine provides low SW computing and high performance cryptographic operation (such as authentication, encryption and decryption). In other words, it's more secure, faster and saves more CPU and Memory resources than software cryptographic operation.

### 14.2. Feature list

- Supported authentication algorithms:
  - MD5
  - SHA-1
  - SHA-2 (SHA-224 / SHA-256)
  - HMAC-MD5
  - HMAC-SHA1
  - HMAC-SHA2 (SHA-224 / SHA-256)
- Supported Encryption / Decryption mechanisms:
  - AES-128 (CBC / ECB / CTR / CFB / OFB / GCTR / GCM)
  - AES-192 (CBC / ECB / CTR / CFB / OFB / GCTR / GCM)
  - AES-256 (CBC / ECB / CTR / CFB / OFB / GCTR / GCM)
- Supported programmable CRC

# Tuya module

power diagram

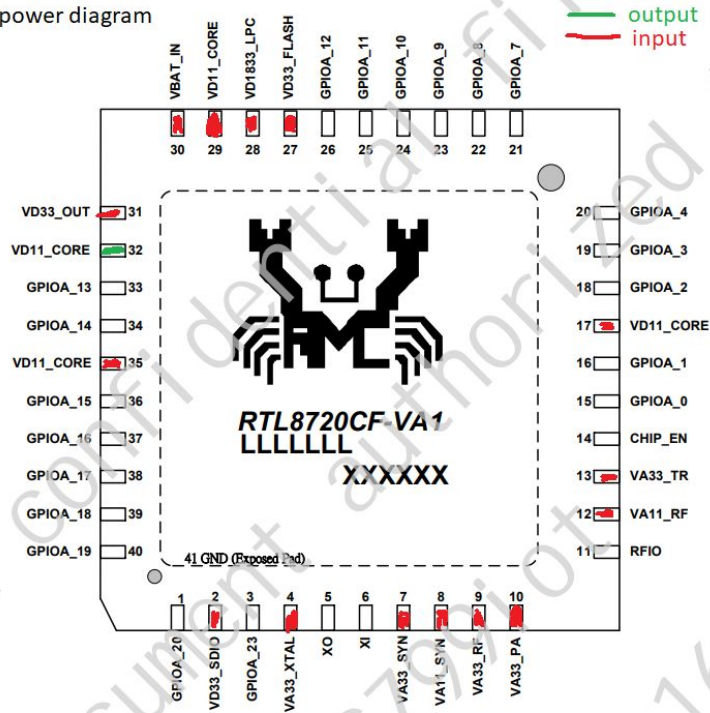


Figure 1 RTL8720CF-VA1 QFN40 Pin Assignments

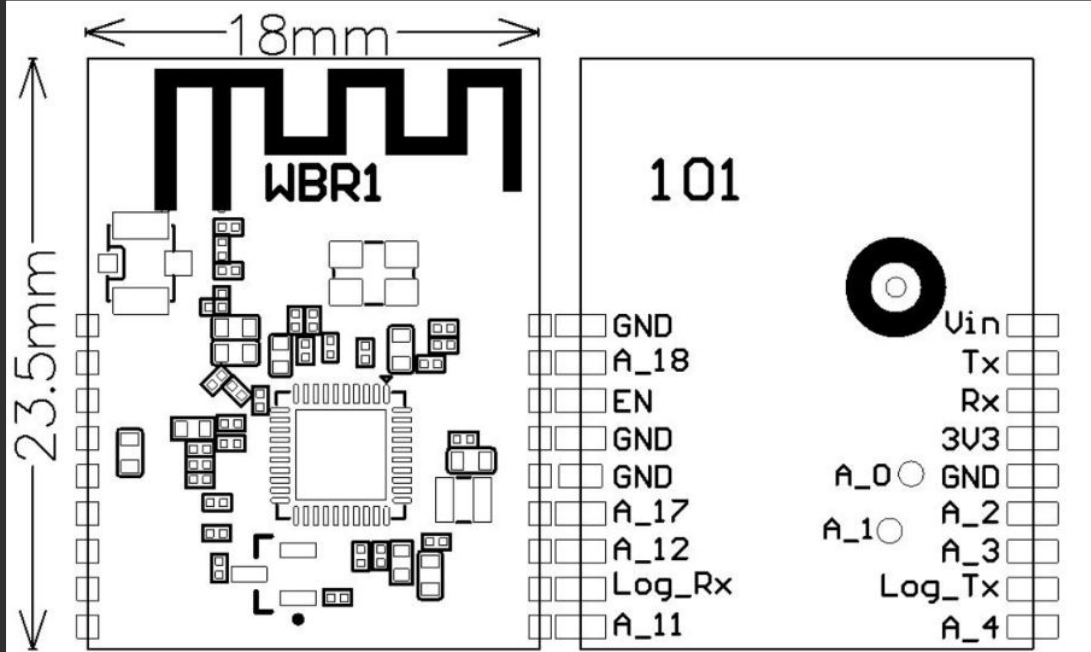
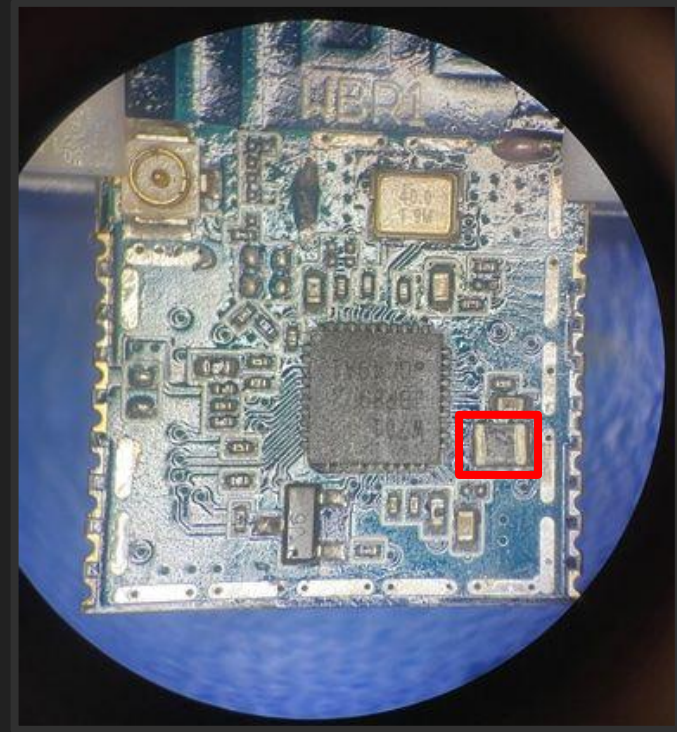
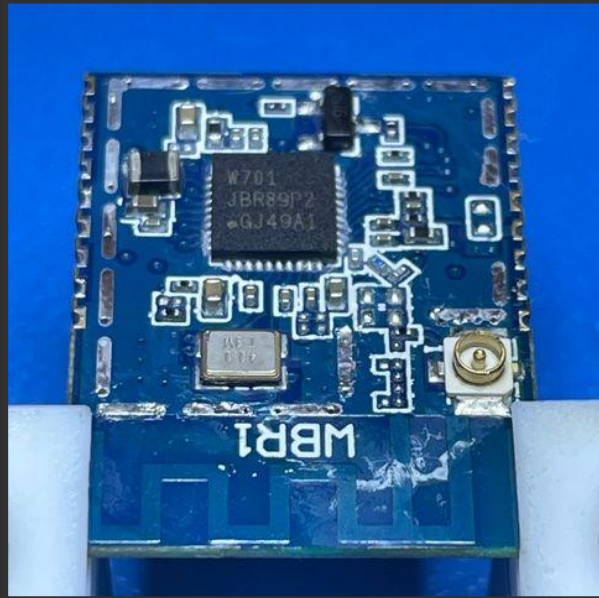


Figure 2-1 WBR1 front and rear views

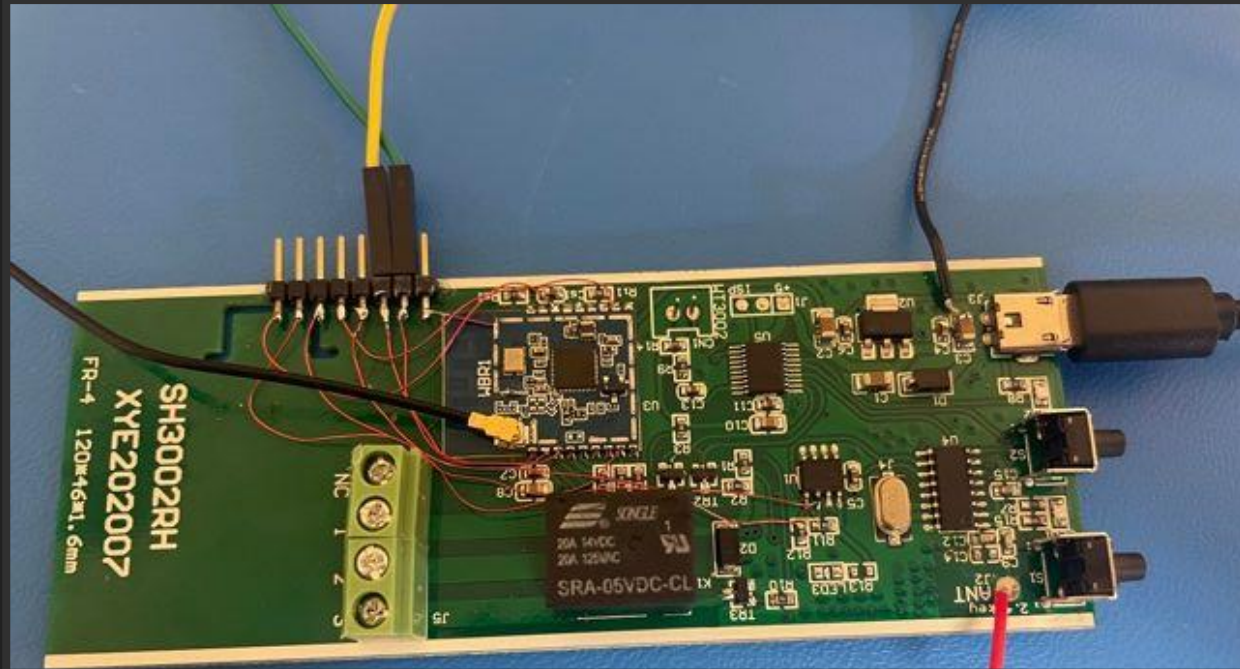
# Tuya module



Overview of the chip, notice the external inductor for the switching supply. This will generate the  $\sim 1.15\text{V}$  core voltage. There is a low amount of DC coupling required for this chip. Notice also the RF matching region, clock region, some external transistor, and various passives.

# Sniffing interesting pins

Lets go right to sniffing this Tuya module. The ISP header was interesting but let's compromise this first.



# UART log and debug mode...

```
== Rtl8710c IoT Platform ==  
Chip VID: 5, Ver: 1  
ROM Version: v2.1
```

```
== Boot Loader ==  
Dec 5 2019:14:02:18
```

```
fwx SELE[ffffffff]  
fw SELE Bitidx 0, fw1 valid 1, sn 100, fw2 valid 1, sn 101  
fw1 USE, return sn 100
```

```
Boot Loader <==  
== RAM Start ==
```

```
Build @ 12:51:08, Jul 29 2020
```

```
Create Task init, stack 0x1000fb28, len 5120  
Create Task app_init, stack 0x10010f88, len 8192  
Create Task IDLE, stack 0x10012fe8, len 768  
Create Task Tmr Svc, stack 0x10013640, len 2048  
Create Task TCP_IP, stack 0x10013fb8, len 4000
```

```
[.....]
```

```
[01-01 18:12:15 TUYA Debug][tuya_device.c:19] < TUYA IOT SDK V:1.0.12 BS:40.00_PT:2.2_LAN:3.3_CAD:1.0.2_CD:1.0.0 >  
< BUILD AT:2020_07_28_19_56_28 BY embed FOR ty_iot_wf_bt_sdk_rtos AT rtl8720cf_ameba >  
IOT DEFS < WIFI_GW:1 DEBUG:1 KV_FILE:0 SHUTDOWN_MODE:0 LITTLE_END:1 TLS_MODE:2 ENABLE_LOCAL_LINKAGE:0 ENABLE_CLOUD_OPERATION:0  
ENABLE_SUBDEVICE:0 ENABLE_ENGINEER_TO_NORMAL:0 OPERATING_SYSTEM:2 ENABLE_SYS_RPC:0 TY_SECURITY_CHIP:0  
RELIABLE_TRANSFER:RELIABLE_TRANSFER ENABLE_LAN_ENCRYPTION:1 ENABLE_LAN_LINKAGE:0 ENABLE_LAN_LINKAGE_MASTER:0 >
```

```
[.....]
```

```
[01-01 18:12:15 TUYA Debug][tuya_device.c:20] rtl8720cf_common_user_config_ty:2.1.6
```

```
[.....]
```

```
[01-01 18:12:15 TUYA Notice][simple_flash.c:498] get key:
```

```
0x30 0xc6 0x16 0x10 0xb2 0x43 0xb 0xf1 0x73 0xf1 0x3 0x3b 0x71 0x3 0x51 0x67
```



# The other UART

This communicates with 5V MCU using transistors as level shifters and a few resistors. If the WBR1 module was in circuit, the other MCU would talk to it over UART. If it is out of circuit, it would wait for UART communication trying different bit rates.

The format is all on Tuya's website for developers. We could add a hardware implant here.

```
[01-01 18:12:15 TUYA Notice][tuya_uart_common_api.c:118] ty_uart_common_main_components_version:1.0.5,memrory left:67400...  
[01-01 18:12:15 TUYA Notice][tuya_uart_adapt.c:421] ty_uart_public_auto_adapt_component_version:1.0.1
```

```
Create Task uart_adapt_task, stack 0x1002e850, len 2048  
[01-01 18:12:15 TUYA Notice][tuya_uart_adapt.c:358] try:9600/3/1/0  
[01-01 18:12:15 TUYA Notice][tuya_uart.c:134] 1 9600  
[01-01 18:12:18 TUYA Notice][tuya_uart_adapt.c:358] try:9600/3/1/0  
[01-01 18:12:18 TUYA Notice][tuya_uart.c:134] 1 9600  
[01-01 18:12:22 TUYA Notice][tuya_uart_adapt.c:358] try:9600/3/1/0  
[01-01 18:12:22 TUYA Notice][tuya_uart.c:134] 1 9600  
[01-01 18:12:25 TUYA Notice][tuya_uart_adapt.c:358] try:115200/3/1/0
```

[.....]

Durablowl State On

- Tuya: 55 AA 00 00 00 00 FF
- MCU: 55 AA 03 00 00 01 04 07

[...]

[MCU sends more packets]

Durablowl State Off

- Tuya: 55 AA 00 00 00 00 FF
- MCU: 55 AA 03 00 00 01 06 09

[stops sending packets]

## Frame format description

Field	Length (byte)	Description
Header	2	It is fixed as 0x55aa
Version	1	It is used for upgrade and extension
Command	1	Specific frame type
Data length	2	Big-endian
Data	N	Entity data
Checksum	1	Start from the header, add up all the bytes, and then divide the sum by 256 to get the remainder

# Datasheet finding... debug mode? SWD/JTAG?

## 1.3.4.1 Power On Trap Pin

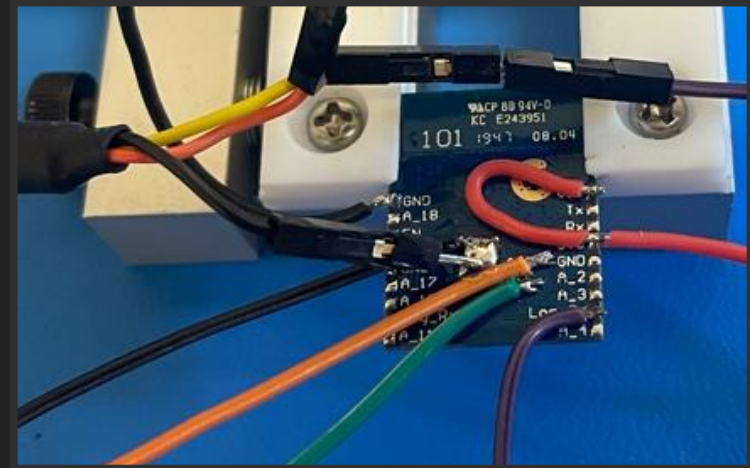
Table 3 Power On Trap Pins

Symbol	Type	RTL8720CF-VA1	RTL8720CM-VA1	RTL8720CN-VA1	Description
TEST_MODE_SEL	I	15	15	15	Shared with GPIOA_0 1: Enter into test/debug mode 0: Normal operation mode
Autoload_Fail	I	16	16	16	Shared with GPIOA_1 1: eFUSE settings are not loaded 0: eFUSE settings are loaded
SPS_LDO_SEL	I	3	3	3	Shared with GPIOA_23 1: LDO 0: SWR

# Trying to get into SWD/JTAG

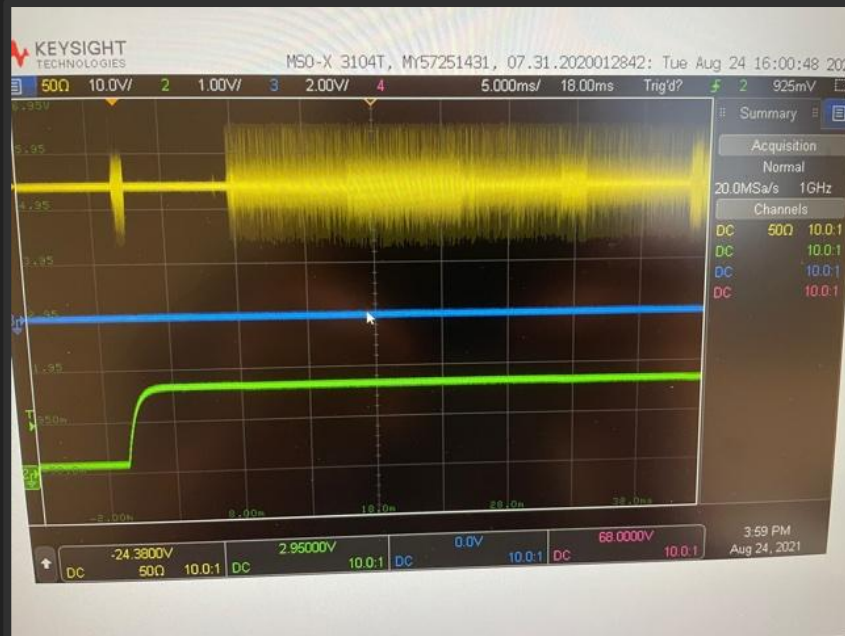
I own a real JLink :P and an XDS110, but also FT2232H, bus pirate, etc

There shouldn't be an issue if there is SWD/JTAG/cJTAG, but I was unable to get this working. I think there is an eFuse that controls access overriding the strap pin? Maybe I can glitch into it? I would have came back to this if not for other attacks.





# Side channel profile on reset



You can see the chip has variation in the electromagnetic signature over time. It could be that during the period where the main processor core is off, it is computing some cryptographic hash or decrypting something with the hardware engine and the main processor is waiting to be interrupted. There is some variation in currents flowing during the flat region still though.

Then the second block of code running continuously is after that (flash?). If we were targeting to glitch some code in the ROM it would be in the beginning of either of these blocks. These gives us some idea of timing operations.

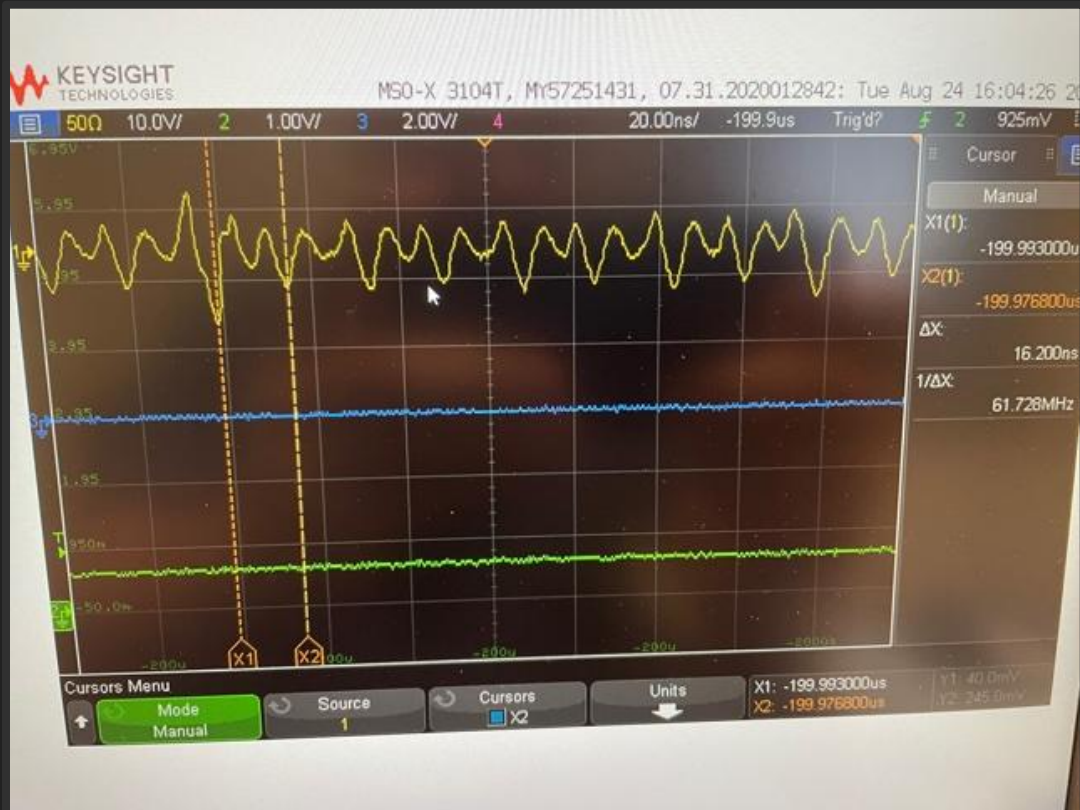
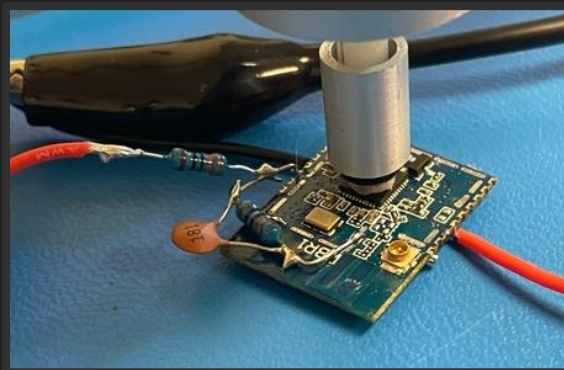
NRST/enable net had a lot of capacitance (rise time long, it has internal pull up), telling me they might have had problems with noise on reset line causing misbehavior. This could be solved by adding capacitance externally or some sort of schmitt trigger input inside the chip (or clean relay delay generator circuit). The circuit may be susceptible to reset glitches.

# Side channel profile largest component

Related to clock?

Variation in features  
is pronounced.

Good for SPA.



# A hypothesis has a chance of being true...

Are there 8 possible test modes? What do they have in this ROM... I wonder? Notice the devs didn't even change the part number from RTL8710 to RTL8720. There are typos in strings all over the ROM/firmware. Both from Realtek and Tuya.

```
== Rtl8710c IoT Platform ==  
Chip VID: 5, Ver: 1  
ROM Version: v2.1  
Test Mode: boot_cfg1=0x20  
Test Mode GPIOA[14, 3, 2] = 0x4
```

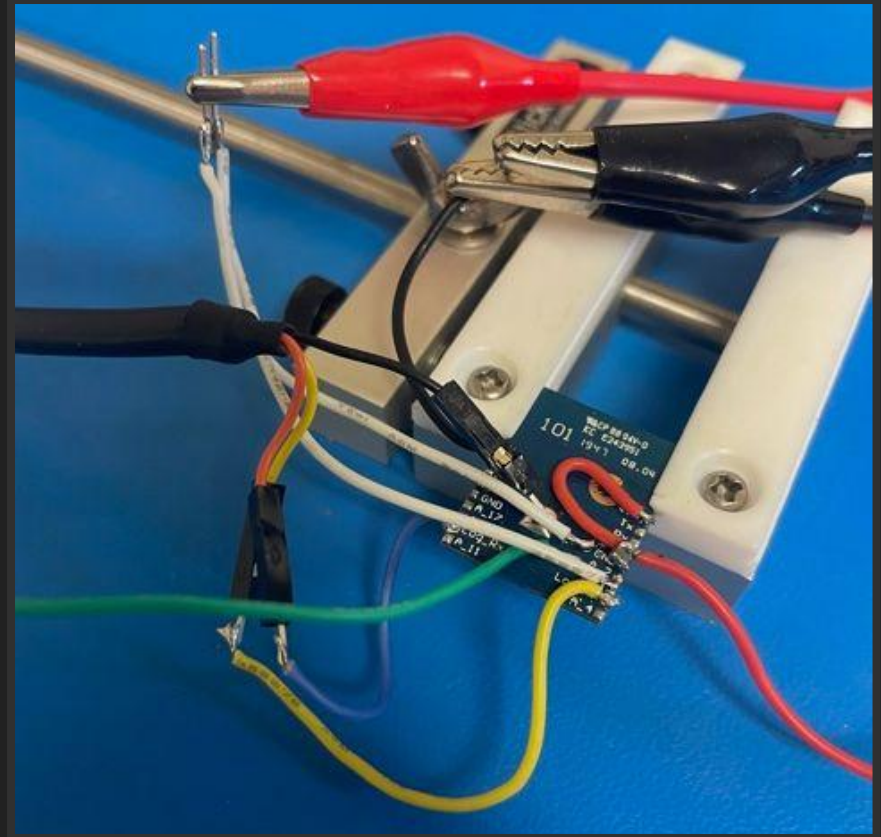
# Playing with on strapping pins

I started searching... and messing with the power pin, GND and 3.3V wire, pulling some pins high and messing with ones I had attached to the JTAG pins after I tried SWD. I eventually figured out what was the internal pin numbers referred to by the ROM UART messages.

# ROM debug menu!

```
== Rtl8710c IoT Platform ==  
Chip VID: 5, Ver: 1  
ROM Version: v2.1  
Test Mode: boot_cfg1=0x20  
Test Mode GPIOA[14, 3, 2] = 0x6
```

```
$8710c>  
$8710c>  
$8710c>?  
DB  
    DB <Address, Hex> <Len, Dec>:  
    Dump memory byte or Read Hw byte register  
  
DHW  
    DHW <Address, Hex> <Len, Dec>:  
    Dump memory half-word or Read Hw half-word register;  
  
DW  
    DW <Address, Hex> <Len, Dec>:  
    Dump memory word or Read Hw word register;  
  
EB  
    EB <Address, Hex> <Value, Hex>:  
    Write memory byte or Write Hw byte register  
    Supports multiple byte writing by a single command  
    Ex: EB Address Value0 Value1  
  
EW  
    EW <Address, Hex> <Value, Hex>:  
    Write memory word or Write Hw word register  
    Supports multiple word writing by a single command  
    Ex: EW Address Value0 Value1  
  
WDTRST  
    WDTRST:  
    To trigger a reset by WDT timeout
```



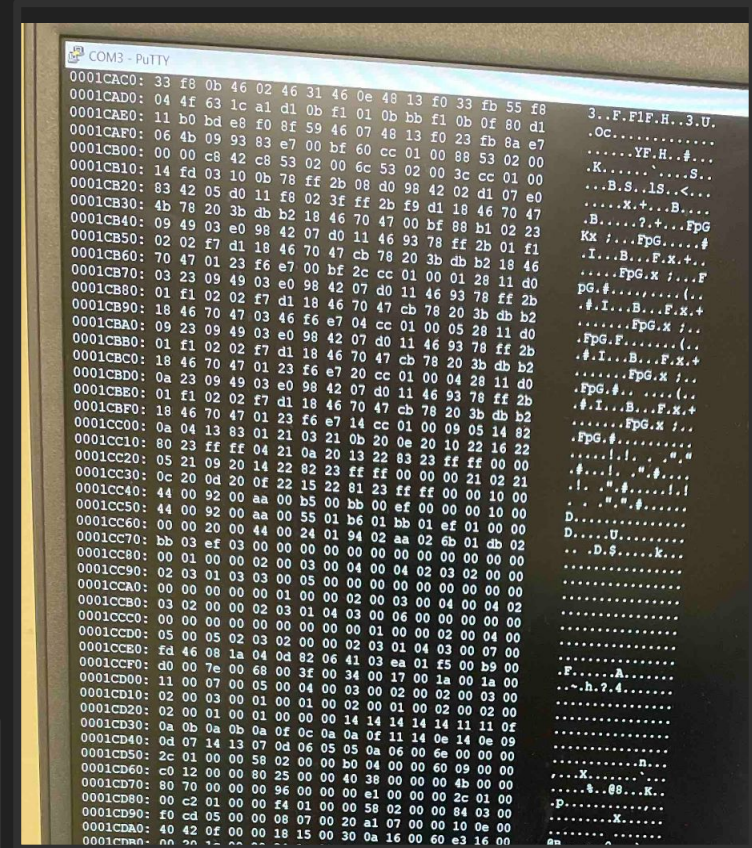


# Thanks for the feature!

Now I have the ROM  
and can do some more  
reverse engineering.

The flash is not  
mapped though so lets  
get code execution and  
map/dump it.

```
piVar6 = (int *)flash_init(0x1002a0e0,uVar19,uVar17);
if (piVar6 != (int *)0x0) {
    uVar4 = 0;
    _DAT_100000ac = (int *)0x0;
    if ((int)(_DAT_100000c4 << 0x1f) < 0) {
        printf("\r[BOOT Err]Flash init error (io_mod=%u, pin_sel=%u)\r\n",uVar19,uVar17);
        goto LAB_00001b2e;
    }
}
```



# So much for "secure" boot

Now I have all the keys used in decryption of flash header (even though flash is internal on this part) and for HMAC SHA authentication. They should have used asymmetric schemes and locked out access to keys after use, or before entering a debug menu that shouldn't exist.

```
15 LAB_00002bfe:
16     get_super_sec_key(_DAT_400000e8,&DAT_1002dd20,0);
17     memcpy(0x1002dd40,&header_key,0x10);
18     memcpy(0x1002dd60,0x98000020,0x40);
19     if ((int)((uint)DAT_40000039 << 0x19) < 0) {
20         FUN_0000d540(0x1002ddc0,0x21,&DAT_1002dd20,0x20);
21         iVar3 = crypto(0x1002ddc0,0x21,0x1002dd60,0x20,0x1002dd40,0x10,0,0);
22         if (iVar3 != 0) goto LAB_00002cd4;
23     }
24     else {
25         FUN_00056f3c(&DAT_1002dda0,&DAT_1002dd20,0x1002dd60);
26     }
27     FUN_000027d4(0x1002dd60);
28     FUN_0000d540(0x1002ddc0,0x21);
29     if (-1 < (char)bVar1) goto LAB_00002bca;
30     iVar3 = crypto(0x1002ddc0,0x21,&DAT_98000060,0x60,0x1002dd40,0x10,0,0);
31     if (iVar3 != 0) {
32         if (_DAT_100000c4 << 0x1f < 0) {
33             printf("\r[BOOT Err]Parttiton Table Header Decry Err!\r\n");
34             return 0xffffffff;
35         }
36         return 0xffffffff;
37     }
```

# So much for "secure" boot

```
1 FUN_00009ab0(_DAT_400000e8,0x1002e2e0,0,0x20,0);
2 copy_setup(0x1002ddc0,0x21,&super_secret_key,0x20);
3 iVar3 = crypto(0x1002ddc0,0x21,0x1002e2e0,0x20,0x1002dd40,0x10,0,0,
4 if (iVar3 != 0) {
5     if (-1 < _DAT_100000c4 << 0x1f) {
6         return 0xffffffff;
7     }
8     printf("\r[BOOT Err]Hash Priv Key Decrypt Err!\r\n");
9     return 0xffffffff;
10 }
11 if ((int)((uint)DAT_40000039 << 0x19) < 0) {
12     copy_setup(0x1002ddc0,0x21,0x1002e300,0x20);
13     iVar3 = crypto(0x1002ddc0,0x21,0x1002dd80,0x20,0x1002dd40,0x10,0,
14     if (iVar3 != 0) {
15 LAB_00002cd4:
16     if (-1 < _DAT_100000c4 << 0x1f) {
17         return 0xffffffff;
18     }
19     printf("\r[BOOT Err]NSK-AES key decrypt Err!\r\n");
20     return 0xffffffff;
21 }
22 }
```

```
FUN_0000e698(0x1002ddc0,0x25);
iVar3 = memcmp(0x1002e320,0x1002e340,0x20);
if (iVar3 != 0) {
    if (_DAT_100000c4 << 0x1f < 0) {
        printf("\r[BOOT Err]Boot Img Hash Result Err!\r\n");
    }
    FUN_00010dbc(&DAT_1002e540,0,0x3a0);
    FUN_0000a81c(&DAT_1002e540,0x3a0);
    return 0xffffffff;
}
```



# Code execution from debug menu

After reverse engineering the ROM... I find the command handler for the debug menu. It is setup so that a variable in SRAM points to an area in ROM that has pointers to the command string, help menu string, and a function pointer to an implementation. I cloned this structure into another region of unused SRAM and added my own command named "golden." I also have python scripts quickly made up to instrument the ROM debug menu for my purposes.

Notes:

0x1002f050 command array pointer

0x1002dd08 flash\_already\_init

0x20000000 32kb free sram

0x318d0 command array in the rom  
length is 84 + the 4 zero bytes at the end

structure:

0: str pointer

4: func addr

8: help string

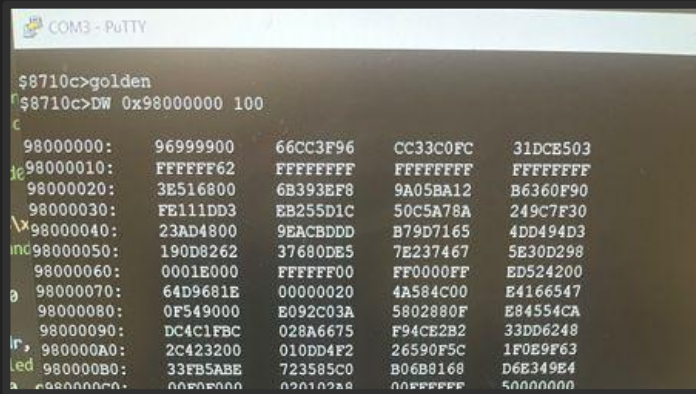
```
command_struct = b'\x18\x93\x05\x00q\x18\x03\x00\x1c\x93\x05\x001\x93\x05\x00\x16\x03\x00p\x93\x05\x00\xcc\x
#command_struct = command_struct[:len(command_struct) - 12] # chop off last command

cmdstraddr = 0x20000200
write_string(cmdstraddr, "golden")
# shell code is compiled for base address 0x10000000
write_bytes(0x10000000, open("shellcode/output.bin", "rb").read())
new_struct = command_struct + struct.pack("<LLL", cmdstraddr, 0x10000000 + 1, cmdstraddr) + b"\x00\x00\x00\x00"
write_bytes(0x20000000, new_struct)
write_word(0x1002f050, 0x20000000)

# golden
# DW 0x98000000 100
```

# Mapping the flash and dumping it

I first map the flash, then call some function that verifies the flash and populates a structure with the pointer to the code entrypoint. We then write some quicker dumping code that writes the raw bytes over the UART console. It is a 2MB flash so this is faster than the read memory commands in the ROM.



```
COM3 - PuTTY
$8710c>golden
$8710c>DW 0x98000000 100
c
98000000: 96999900 66CC3F96 CC33C0FC 31DCE503
98000010: FFFFFFF6 FFFFFFFF FFFFFFFF FFFFFFFF
98000020: 3E516800 6B393EF8 9A05BA12 B6360F90
98000030: FE111DD3 EB255D1C 50C5A78A 249C7F30
98000040: 23AD4800 9EACBDDD B79D7165 4DD494D3
98000050: 190D8262 37680DE5 7E237467 5E30D298
98000060: 0001E000 FFFFFFF0 FF0000FF ED524200
98000070: 64D9681E 00000020 4A584C00 E4166547
98000080: 0F549000 E092C03A 5802880F E84554CA
98000090: DC4C1FBC 028A6675 F94CE2B2 33DD6248
980000A0: 2C423200 010DD4F2 26590F5C 1F0E9F63
980000B0: 33FB5ABE 723585C0 B06B8168 D6E349E4
980000C0: 00000000 00000000 00000000 00000000
```

```
push { r0, r1, r2, r6, r7, lr }

# actually flash_init 0x29bc
ldr r7, =0x29bd

# pin_sel
ldr r2, =1
# io_mod
ldr r1, =0

ldr r0, =0x1002a0e0
blx r7

# lets try to init this crypto stuff
# flash_boot 0x3ad4
ldr r7, =0x3ad5
ldr r0, =0x100000ac
blx r7

# dump this structure?
# ldr r6, =0x100000ac

# dump the flash
# putchar 0x11629 this is for our uart
ldr r7, =0x11629
ldr r6, =0x98000000
.LOOP:
ldrb r1, [r6]
ldr r0, =0
blx r7

add r6, r6, #1
b .LOOP

pop { r0, r1, r2, r6, r7, pc }
```

# Tuya firmware

Now I have the Tuya firmware and ROM. The firmware is not designed too well, it seems to support many features not used in the product. It seems to be a portable firmware that takes in a JSON configuration from Tuya to configure the device?

We could now backdoor the firmware. Since all the keys are known, I can generate a new HMAC signature for the firmware. There is a debug menu mode for UART flash download, but I can also write to the flash via ROM commands.

# Interesting glitches

This device is quite unstable, by just playing with the power pin manually, I can generate interesting glitches and subsequence output over the UART log.

```
== Rtl8710c IoT Platform ==  
Chip VID: 5, Ver: 1  
ROM Version: v2.1  
[SPIF Err]Invalid ID  
[SPIF Err]Invalid ID  
[BOOT Err]Flash init error (io_mod=0,  
pin_sel=0)  
StartUp@0x0: Invalid RAM Img Signature!
```

# Interesting glitches

```
OM Version: v2.1
Test Mode: boot_cfgc\x5
== Rtl8710c IoT Platform ==
Chip VID: 5, Ver: 1
R5rj
== Boot Loader ==
Dec  5 2019:14:02:18
```

```
fwx SELE[ffffffff]
fw SELE Bitidx 0, fw1 valid 1, sn 100,
fw2 valid 1, sn 101
fw1 USE, return sn 100
[MISC Err]Hash Result Incorrect!
Boot Load Err!
```

This output supports our theory that the large flat low power region in the side channel overview capture is the hash function executing over the whole flash. As this takes a good bit of time and would be easier to corrupt/hit in time.

# Interesting glitches

```
Bus Fault:
SCB Configurable Fault Status Reg = 0x00000400

Bus Fault Status:
BusFault Address Reg is invalid(Asyn. BusFault)
Imprecise data bus error:
a data bus error has occurred, but the return address in the stack
frame is not related to the instruction that caused the error.
```

```
S-domain exception from Thread mode, Standard Stack frame on S-PSP
Registers Saved to stack
```

```
Stacked:
R0 = 0x0002fe57
[.....]
R12 = 0x0000001c
LR = 0x10001eaf
PC = 0x00006114
PSR = 0x61000000
```

```
Current:
LR = 0xffffffff
[.....]
SVC priority: 0x00
PendSVC priority: 0xe0
Systick priority: 0xe0
```

```
MSP Data:
1003F9E0: 00000000 E000E014 E000E010 100005EC
1003F9F0: 0000001C 9B01342F 9B012FB8 [.....]
1003FAC0: 04112ED7 FF5428A7 2A42A634 ABF5CAAF
1003FAD0: 564A8902 9576CD37 4461A974 B6ED2E97
```

```
PSP Data:
10012BA8: 0002FE57 0000006F ABDCF628 DEADBEEF
10012BB8: 0000001C 10001EAF 00006114 61000000
10012BC8: 0000006F 00000003 00000000
[.....]
10012C88: 10015658 9B03A4C1 00000000 9B01678F
10012C98: 00000004 10015E88 00001000 9B05E363
```

```
== NS Dump ==
CFSR_NS = 0x00000000
[.....]
PSF_NS = 0x00000000
NS HardFault Status Reg = 0x00000000
SCB Configurable Fault Status Reg = 0x00000000
```

```
== Back Trace ==

msp=0x1003f9e0 psp=0x10012ba8
Process stack back trace:
top=0x10013ba8 lim=0x10012ba8
00006114 @ sp = 00000000
[.....]
00030131 @ sp = 10012c28
9b05e171 @ sp = 10012c34
```

```
Backtrace information may not correct! Use this command to get C source
level in formation:
arm-none-eabi-addr2line -e ELF_file -a -f 00006114 10001eab 0002eaa5
0002fe53 9b 00fef9 9b00fef9 00030131 9b05e171
```

I guess this chip just loves to be unstable but keep running. Should be able to glitch it if debug mode was not enabled with some limited setup...

# Ah that is boring...

Let's do something more fun!

- Lets see if we can glitch into the debug menu without messing with pins?
- Or encrypt our own new firmware and bypass the HMAC check?
- Overflow UART receive message length to get buffer overflow in firmware?

Many things are possible at this point. The device ROM was actually designed such that it uses an external SPI flash for non-RTL8720CF part numbers... so a correlation power analysis attack (CPA) would be fun to grab the header key then break secure boot because they use an HMAC SHA for authenticity.

# Back to the WiFi/Bluetooth communication

Now that I have the firmware to the Tuya module. I can reverse engineer communication with the backend/cloud, I have the TLS pre-shared key. I can decipher the structures of data on LAN, and send the device my own messages. It is all a matter of time at this point.

Keys are fun! I could have gotten some from the App, but this is better! There may be some provisioned ephemeral keys, but they are not per-session. It would be setup when the device is linked to the Tuya account/app.

Cipher Suites (4 suites)

```
Cipher Suite: TLS_PSK_WITH_AES_256_CBC_SHA (0x008d)
Cipher Suite: TLS_PSK_WITH_AES_128_CBC_SHA256 (0x00ae)
Cipher Suite: TLS_PSK_WITH_AES_128_CBC_SHA (0x008c)
Cipher Suite: TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0x00ff)
```



Questions?