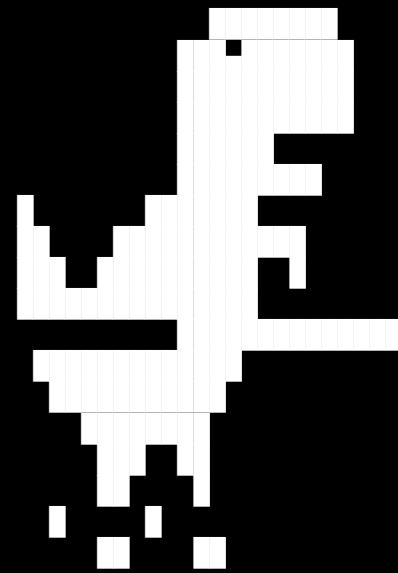


R2eDN

◦—2—◦—0—◦—2—◦—0

# ESIL SIDE-CHANNEL SIMULATION



Glitch0z0r 3000

# > iq

- Nicolas
- R&D dept. @ Kudelski Security
- Embedded systems security research
  - Reverse engineering
- Karim & Sylvain
- Security Evaluation Lab @ Kudelski IoT
- Hardware attacks
  - Glitch / EM
  - Lasers !



# Introduction

- Fault attack is an interesting attack path for IoT
- For low cost devices, there is only software protection
- Set-up cost have decreased a lot recently

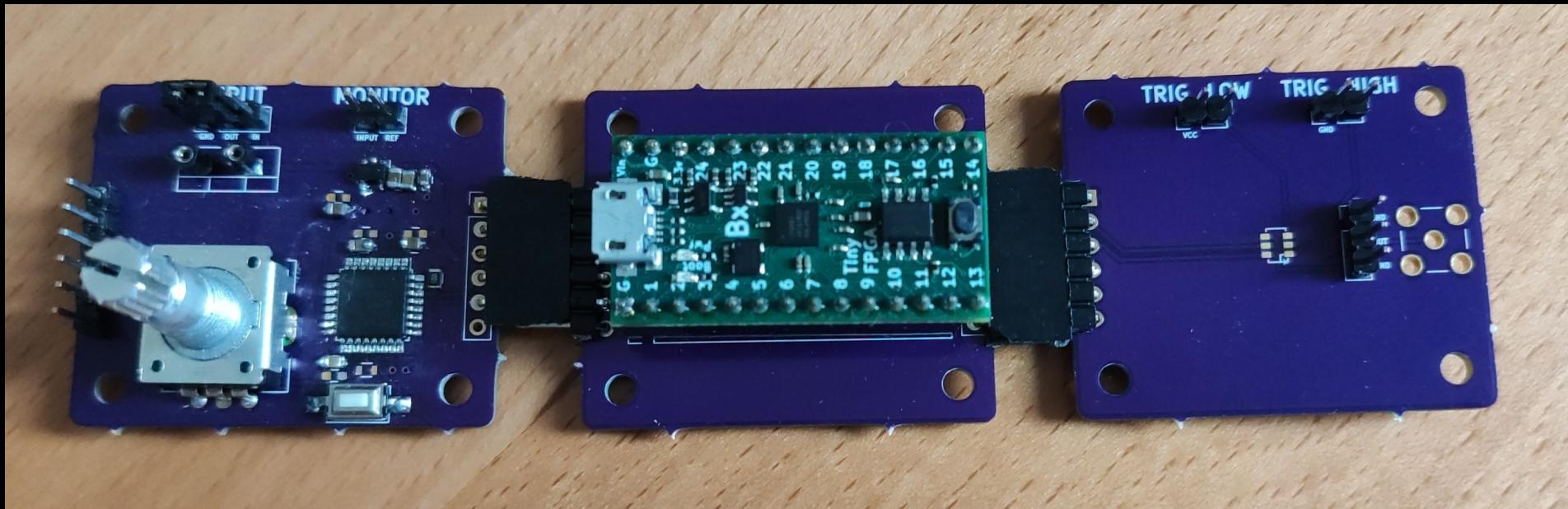


# FAULT ATTACKS

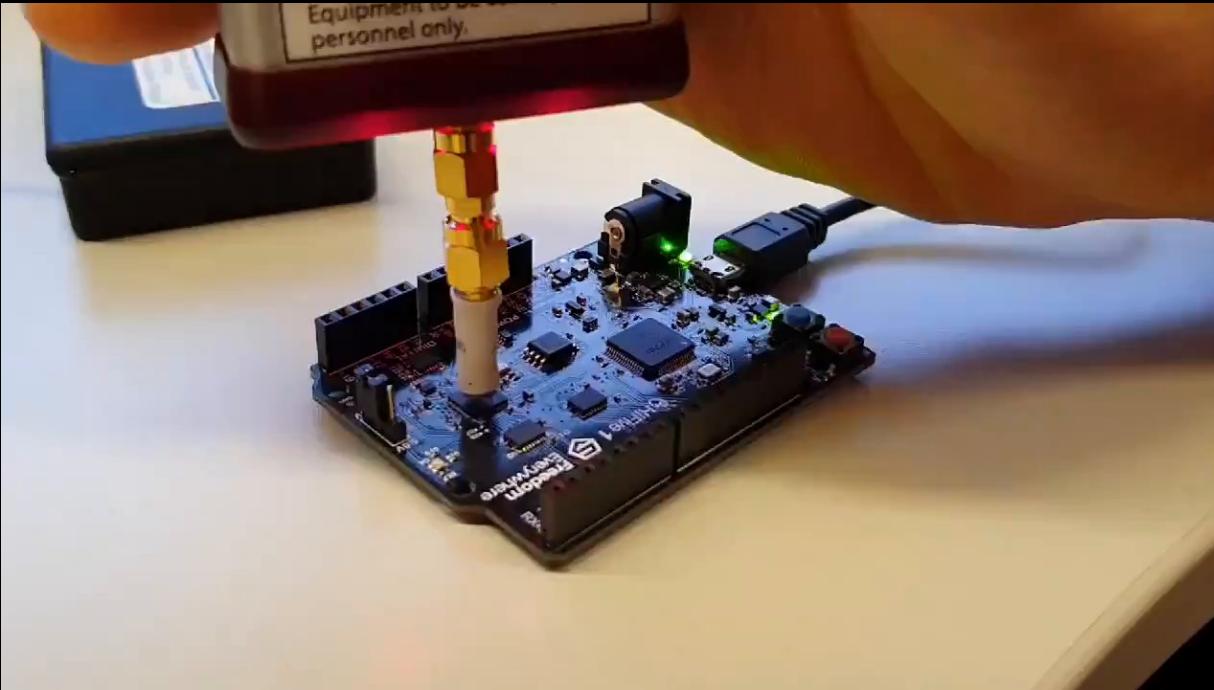
# Fault attacks

- Disturb a device execution from its normal behavior
- Various hardware technique allows that
- Some software or micro-architectural methods as well:  
Rowhammer, LVI, VOLTPwn, ...

# Fault attacks



# Fault attacks



# Fault attacks



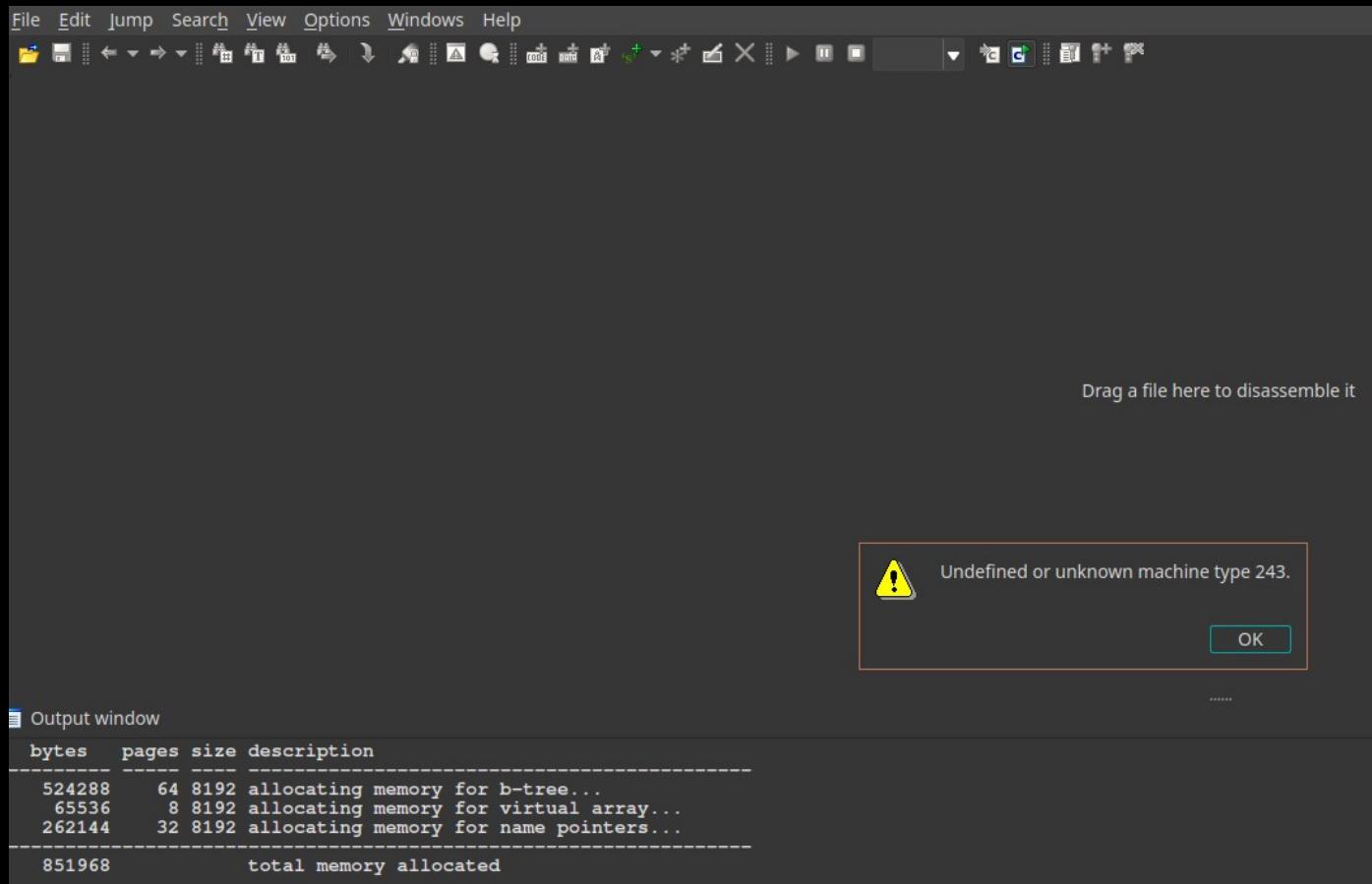
# Fault effects

- Effects on the CPU can vary in many ways
  - Skip instruction(s)
  - Corrupt register/memory reads/writes
- Literally depends on the silicon, the type of fault, ...

# Simulate fault effects

- Glitching is a first step but finding practical exploits is another
- Not always access to source code
- Not always easy to emulate a firmware
- Hardware glitch campaigns may be very long
- Useful for code audit to catch some vulnerabilities

# Problems ?



# Existing tools

- **Lazart** (VERIMAG laboratory, University of Grenoble)
  - Based on LLVM
  - Needs source :(
- **FiSim** (Riscure)
  - Based on Unicorn & Capstone
  - Only works on supported architectures :(
  - Not publicly available :(



## ESIL FOR FAULT SIMULATION

# Our approach

- Use R2's ESIL to simulate parts of the firmware
  - On raw firmware, can be used during the reversing phase with a single tool
- Instrument the VM using r2pipe and Python
  - Setup stack/registers
  - Run the simulation
  - Record the results

# GLitchoz0r 3k

- Python module
- Easy setup
  - Set simulation start/end address
  - Set initial registers
  - Set success conditions
  - ...
  - Results !

```
import glitchoz0r3k

glitchoz0r3k.GLITCHES = [glitchoz0r3k.glitch.Skip]

g = glitchoz0r3k.Glitchozor()

g.open('testcases/target')
g.set_start(0x000012da)
g.set_end(0x00001327)
ops = g.analyze()
print(f"Number of ops : {ops}")

def conditions(ctx):
    """
    Tests for registers to verify that our glitch worked
    """
    if ctx['regs']['rax'] == 0:
        return True
    else:
        return False

g.set_conditions(conditions)

for r in g.run():
    print(r['glitch_str'])
```

# Fault models and conditions

- All fault models are implemented as a function
- Fault models are added to the simulation
- One fault model per simulation

```
class Skip(Glitch):
    """
    Simple glitch. Skips the instruction
    """

    def apply(pipe):
        cur_instr = pipe.cmdj("pdj 1@PC")[0]
        new_addr = cur_instr['offset']+cur_instr['size']
        pipe.cmd('aer PC = '+hex(new_addr))
        return f"Skip {cur_instr['disasm']} @ {hex(cur_instr['offset'])}"
```

```
class ZeroDReg(Glitch):
    """
    Zero destination register
    """

    def apply(pipe):
        cur_instr = pipe.cmdj("pdj 1@PC")[0]
        changes = pipe.cmdj('aeaj 1@PC')
        try:
            reg = changes['W'][0]
            pipe.cmd(f'aer {reg} = 0')
            return f"Zero {reg} in {cur_instr['disasm']} @ {hex(cur_instr['offset'])}"
        except:
            return None
```



# RESULTS

# Naive check

```
1 #include "stdio.h"
2 #include "string.h"
3 #include "stdint.h"
4
5 uint8_t check1(uint8_t * src, uint8_t * dst)
6 {
7     uint8_t i = 0, res = 0;
8     for(i=0; i<8; i++) {
9         res |= (src[i] ^ dst[i]);
10    }
11    return res;
12 }
13
14 int main(int argc, char * argv[])
15 {
16     uint8_t source[8] = {1,2,4,0,1,8,1,2};
17     uint8_t destination[8] = {1,2,3,4,5,6,7,8};
18
19     if(check1(source, destination)) {
20         printf("Bad boy\r\n");
21     } else {
22         printf("Good boy\r\n");
23     }
24 }
```

# Naive check

```
~/Projects/glitch-simulation master*
> python3 t
```

- Skipping the for loop makes the password check return a correct value

```
0x00001184 0fb645fe    movzx eax, byte [i]      ; target.c:9  for(i=0; i<8; i++) {
0x00001188 83c001       add eax, 1
0x0000118b 8845fe       mov byte [i], al
; CODE XREF from dbg.check1 @ 0x1161
0x0000118e 807dfe07    cmp byte [i], 7
0x00001192 76cf        jbe 0x1163
0x00001194 0fb645ff    movzx eax, byte [res]   ; target.c:12  return res;
0x00001198 5d           pop rbp                 ; target.c:13 }
```

# Stronger check

```
uint8_t check2(uint8_t * src, uint8_t * dst)
{
    uint8_t i = 0, res1 = 0, res2 = 0;
    for(i=0; i<8; i++) {
        res1 |= src[i] ^ dst[i];
    }

    for(i=0; i<8; i++) {
        res2 |= src[i] ^ dst[i];
    }

    if (res1 != 0 || res2 != 0) {
        return 1;
    } else {
        return 0;
    }
}
```

# Stronger check

- If statement is a branch
  - Branches follow each other

```
> python3 target_glitch.py
Number of ops : 278
100%|██████████| 278/278 [00:11<00:00, 23.98it/s]
Skip movabs rax, 0x807060504030201 @ 0x1306(count=10)
Skip jmp 0x1238 @ 0x1231(count=275)
```

```
; CODE XREFS from dbg.check2 @ 0x119a, 0x11ed
0x0000121a    807dfd07      cmp byte [i], 7
0x0000121e    76cf         jbe 0x11ef
0x00001220    807dfe00      cmp byte [res1], 0          ; target.c:27      if (res1 != 0 || res2 != 0)
0x00001224    7506         jne 0x122c
0x00001226    807dff00      cmp byte [res2], 0
0x0000122a    7407         je 0x1233
; CODE XREF from dbg.check2 @ 0x1224
0x0000122c    b801000000      mov eax, 1          ; target.c:29      return 1;
0x00001231    eb05         jmp 0x1238
; CODE XREF from dbg.check2 @ 0x122a
0x00001233    b800000000      mov eax, 0          ; target.c:33      return 0;
; CODE XREFS from dbg.check2 @ 0x119a, 0x1231
0x00001238    5d             pop rbp           ; target.c:35 }
0x00001239    c3             ret
```

# Stronger stronger check

```
uint8_t check3(uint8_t * src, uint8_t * dst)
{
    uint8_t i = 0, res1 = 0, res2 = 0;
    for(i=0; i<8; i++) {
        res1 |= src[i] ^ dst[i];
    }

    for(i=0; i<8; i++) {
        res2 |= src[i] ^ dst[i];
    }

    if (res1 == 0 && res2 == 0) {
        return 0;
    } else {
        return 1;
    }
}
```

# Fault attacks on AES

- Well known and documented fault attack
  - Works great on embedded devices
- With at least 4 faulted ciphertexts, it is possible to recover the AES key
- If we can generate good faulted ciphertexts and recover the key, the glitch simulation is working

# The *famous* Balda test

- “*If it simulates AES , ESIL is close enough*”
  - Good example (memory reads/writes, arithmetic operations, loops, ...)
- Problem : ESIL works fine on x86, but does not work on ARM nor RISC-V
  - Incorrect results, or infinite loops

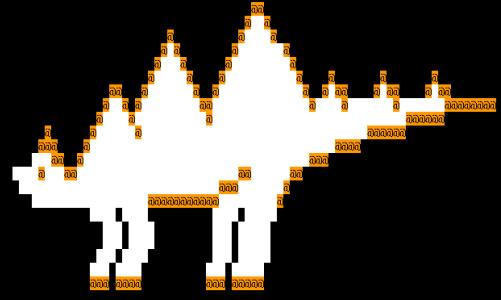
R2CON  
•—2—0—•2—0

## ESIL / ASM DIFFING

# Goals

- Validate ESIL implementation against a real device
- Improve ESIL support for embedded architectures (e.g. ARM)
- Automate comparison process for:
  - Few instructions / functions
  - Full binary execution
- Architecture agnostic

# Diffosaurus



- Instrumentation of Radare2 with Python (r2pipe)
- Simultaneously open two pipes
  - DEBUG
    - QEMU
    - Device [JTAG / SWD]
  - ESIL
    - ESIL VM setup
    - ESIL synchronization with DEBUG (registers & memory space)
- Step and compare registers until discrepancy happens ... or not :)

# Use case

- Compiled binary
  - Specific function within firmware
- Unit tests
  - Relocate current map to executable memory (e.g. SRAM)
    - "omb. 0x20000000"
  - Write assembly
    - "wa MOV r1,#0x0;SUBS r1,#1;SUBS r1,#1;"@ addr
  - Configure “instruction pointer”
  - Execute

External RAM	0×9FFFFFFF
Peripheral	0×60000000
SRAM	0×5FFFFFFF
Test	0×40000000
Code	0×3FFFFFFF
	0×20000000
	0×1FFFFFFF
	0×00000000

# Features

- Display previous and current instruction
- Highlight differences between registers
- Synchronize registers between DEBUG and ESIL pipes
- Interact with DEBUG / ESIL pipes

# Demo

```
ESIL:
[--] name addr      bytes      disasm      comment esil      refs      xrefs
r15 0x000102b4 02f098f9 bl #0x125e8      pc,lr,=,75240,pc,= 0x000125e8 0x000102b8

[PC] name addr      bytes      disasm      comment esil      refs      xrefs
r15 0x000125e8 80b5 push {r7, lr}      8,sp,=,lr,r7,2,sp,=[*]

DEBUG:
[--] name addr      bytes      disasm      comment esil      refs      xrefs
pc 0x000102b4 02f098f9 bl #0x125e8      pc,lr,=,75240,pc,= 0x000125e8 0x000102b8

[PC] name addr      bytes      disasm      comment esil      refs      xrefs
pc 0x000125e8 80b5 push {r7, lr}      8,sp,=,lr,r7,2,sp,=[*]

REG      DEBUG      ESIL
sp      0x407ffd68 0x407ffd68
lr      0x000102b9 0x000102b8 DIFF
pc      0x000125e8 0x000125e8
r0      0x00011e70 0x00011e70
r1      0x00000001 0x00000001
r2      0x407ffd74 0x407ffd74
r3      0x00012ae1 0x00012ae1
r4      0x00000000 0x00000000
r5      0x00000000 0x00000000
r6      0x00000000 0x00000000
r7      0x00000000 0x00000000
r8      0x00000000 0x00000000
r9      0x00000000 0x00000000
r10     0x00070b0c 0x00070b0c
r11     0x00000000 0x00000000
r12     0x00012b51 0x00012b51
cpsr    0x00000030 0x00000030

Ooops something changed!
[d]isplay registers
[e]sil → TARGET
[t]arget → ESIL
[s]tep
[i]nteractive
[v]erbose
[q]uit
```

# Demo

1

# ESIL improvements

- 1 PR in progress
  - Fix ARM IT Block (#17509)
- 7 PRs merged
  - ARM fixes ( #17347 / #17058 )
  - RISC-V fixes ( #17115 / #17078 / #17005 / #16996 / #16962 )
- Multiple issues fixed
- AES now works on ARM32/Thumb/RISC-V



## FINAL TESTS AND CONCLUSION

# Final test

- Apply the fault simulator on AES
- If the faults allow to recover the AES key, it is a win

# Final test results

```
Skip bls 0x8704 @ 0x8760(count=21059)
Skip ldrb r2, [fp, -6] @ 0x8704(count=21060)
Skip add r2, r1, r2 @ 0x8714(count=21064)
Skip add r3, r2, r3 @ 0x8718(count=21065)
Skip ldrb r3, [r3] @ 0x871c(count=21066)
Skip mov r0, r3 @ 0x8720(count=21067)
Skip ldrb r2, [fp, -6] @ 0x8724(count=21068)
Skip ldrb r3, [fp, -5] @ 0x8728(count=21069)
Skip ldr r1, [pc, 0x5c] @ 0x872c(count=21070)
Skip ldrb r0, [r1, r0] @ 0x8730(count=21071)
Skip ldr r1, [fp, -0x10] @ 0x8734(count=21072)
Skip add r2, r1, r2 @ 0x873c(count=21074)
Skip add r3, r2, r3 @ 0x8740(count=21075)
Skip mov r2, r0 @ 0x8744(count=21076)
Skip strb r2, [r3] @ 0x8748(count=21077)
Skip ldrb r3, [fp, -6] @ 0x874c(count=21078)
Skip add r3, r3, 1 @ 0x8750(count=21079)
```

8%:  \*

```
4a614e376613ece8918c05ef907b822d
0bf38842e2d571487b490fe850556adc
30f8558ace8a90d873eeacebe92d481d
30f8558ace8a90d873eeacebe92d481d
ea614c8796061506b384561917cac783
ea614c8796061506b384561917cac783
20d43b67809df10b7cb98d4a93b31eb0
20d43b67809df10b7cb98d4a93b31eb0
ea614c8796061506b384561917cac783
20d43b67809df10b7cb98d4a93b31eb0
20d43b67809df10b7cb98d4a93b31eb0
ea614c8796061506b384561917cac783
20d43b67809df10b7cb98d4a93b31eb0
20d43b67809df10b7cb98d4a93b31eb0
20d43b67809df10b7cb98d4a93b31eb0
20d43b67809df10b7cb98d4a93b31eb0
701277f0bb608870e1225a3bce31c2c7
20d43b67809df10b7cb98d4a93b31eb0
c469174e53d5821a9d0c4b859cc94366
e2f6dfb704a578399be07803fb81f03c
: 79/1000 [01:34<09:44, 1.58it/s]
```

## Future work

- Implement multiple faults during the same simulation
- Add fault models
  - Single bit flip on instruction to simulate laser-induced faults
- Enhance ESIL for other architectures

# Conclusion

- Glitch simulation reproduced on real hardware
- Ability to identify potential fault injection points in firmware
- Radare2 allows to *quickly\** add a new architecture with no change in the simulation code
  - \* If ESIL works correctly



## BONUS : INSTRUCTION TRACING USING ESIL

# Side channel tracing

- Since we can simulate a full AES, we can use ESIL information to retrieve memory reads / writes
  - We can generate / plot a trace of memory accesses
- We can apply a “CPA” attack to recover the key
  - Tools already exist <https://github.com/SideChannelMarvels/Daredevil>

# Side channel tracing

