Characterization of a Cortex-M4 microcontroller with backside optical fault injection

Research Project 1

Jasper Hupkens
Dominika Rusek
05.02.2019
Introduction to the world of fault injection

• Research project at Riscure

• Fault injection techniques introduce faults into a target by controlled environmental changes, in order to alter its intended behavior

• 5 types - clock, voltage, electromagnetic, optical, temperature

• Our focus - optical (laser) fault injection
Why?

• Secure software relies on hardware functioning in the intended way

• You can have the best lock in the world on your door, but if your door is made out of paper, it is useless

• Used e.g in bypassing secure boot of Nintendo consoles
Research question

What is the security impact of injecting laser glitches into an ARM based, Cortex-M4 microcontroller (MCU)?

- How may laser glitches be injected into the MCU so that it results in a fault?
- What are the optimal variables for the laser to introduce glitches in the ARM Cortex-M4 MCU?
- What behavioral changes occur in the MCU when injecting laser glitches?
Device Under Test - Cortex-M4
Test environment

Safety metal box

- Control laser glitching
- Riscure Diode Laser Station
- Send glitch trigger
- Riscure glitch amplifier
- 3.3VCC
- Target Cortex-M4
- Computer

Riscure Spider

Research setup
Test environment

Research setup
Methodology

- Global vs detailed scan
- Several laser parameters
- Color coding of the results:
  - Red/pink – success
  - Green – expected
  - Yellow – mute
  - Orange – reset
  - Cyan – timeout
- Glitch repeatability
Results: Counter increment

- Goal: verify the setup, check if glitches can occur
- Result: 0.012% successful glitches
- Different memory and register operations

Code in C:
```c
GPIOC-&gt;BSRRL = GPIO_Pin_2; //Trigger on
while (payload_len) {
    payload_len--;
    upCounter++;
}
GPIOC-&gt;BSRRH = GPIO_Pin_2; //Trigger off
```

Code in ARM assembly:
```
GPIOC-&gt;BSRRL = GPIO_Pin_2; //Trigger on
4b23 ldr r3, [pc, #140] ; (8004f94 <main+0x364>)
2204 movs r2, #4
831a strh r2, [r3, #24]
while (payload_len) {
e00b b.n 8004f24 <main+0x2f4>
    payload_len--;
f8d7 32f8 ldr.w r3, [r7, #760] ; 0x2f8
3b01 subs r3, #1
f8c7 32f8 str.w r3, [r7, #760] ; 0x2f8
    upCounter++;
f107 0320 add.w r3, r7, #32
681b ldr r3, [r3, #0]
1c5a adds r2, r3, #1
f107 0320 add.w r3, r7, #32
601a str r2, [r3, #0]
```
Results: Counter increment
Results: Bitwise increment

• Goal: setting bits in a byte with a consecutive power of 2

• Result: 36.14% successful glitches

• 0xff: 1111 1111
• 0xfb: 1111 1011
• 0xf7: 1111 0111
Results: Bitwise increment
Results: Register value modification

• Goal: Modify value while in register

• How: Initialize registers with known values

• Result: 1.50% successful glitches

• But we are modifying instructions instead
Results: Register value modification

• Register values:
  • \texttt{r0}: fa ca de 00 \texttt{r6}: de ad be ef \texttt{r4}: ca fe ba be \texttt{r5}: fa ce fe ed

• NOP instruction: \texttt{mov r1, r1}

• MOV transformed into Linear Shift Left (LSL)

• Expected output: 0xfacade00deadbeefcafebabefacefeed
Results: ADD loop

• Goal: Increment a counter to 10,000 using a single instruction

• Instruction: add.w r1, r1 #1 repeated 10,000 times

• Result: 50.77% successful glitches
  • 0xDEADD77F
  • 0Xeadc0789
  • 0x1890
Results: ADD loop
Results: ADD loop (0xdeadd77f)

- Register r0 was first loaded with 0xdeadbeef
- This value now shows up in r1
- Subtract 0x1890 from the result

\[
\begin{align*}
0x\text{deadadd77f} \\
0x00001890 \\
\underline{-} \\
0x\text{deadbeef}
\end{align*}
\]
Results: ADD loop (0xeadc0789)

• The same was true for this result

• When we subtract 0x1890 from result
Results: ADD loop

• So how can this happen?

• We modified the processor instruction, instead loading r1 it loads r0
Results: ADD loop

• How could we obtain the value of 0x1890

• Probably the counter was restarted, also this can be explained using a modified instruction

• The AND instruction sets the counter back to 1 or 0
Bypass authentication

• Goal: Attack a real-world scenario, in this case, password verification

• Result: 0.22% successful glitches

• Lots of possibilities for introducing glitches

```c
volatile int charsOK = 0;
authenticated = 0;
get_bytes(4, rxBuffer);
GPIOC->BSRRL = GPIO_Pin_2; //Trigger on
for (i = 0; i < 4; i++) {
    if (rxBuffer[i] == password[i]) {
        charsOK = charsOK + 1;
    }
}
GPIOC->BSRRH = GPIO_Pin_2; //Trigger off
if (charsOK == 4) {
    authenticated=1;
    send_char(0x90);send_char(0x00);
} else {
    send_char(0x69);send_char(0x86);
} break;
```
Results: Bypass authentication
Conclusion

• There are two ways laser injection can be performed - backside and frontside

• Power 20-25% of the maximum 20W seemed to be most efficient

• Other variables differ per experiment

• We have proven to be able to modify processor instructions

What is the security impact of injecting laser glitches into an ARM based, Cortex-M4 microcontroller (MCU)?
Future work

• Use of different objectives: magnitude 20x or 50x to have smaller spotsize and more precise aim

• Target specific features of the board e.g. the Read Data Protection (RDP) byte

• Test other processors in Cortex family with more advanced security features e.g. TrustZone or Memory Protection Unit (MPU)
Thank you! Questions?