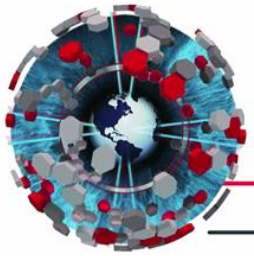


11ª EDIÇÃO 2014

H2HC

HACKERS TO HACKERS CONFERENCE

.inf INSTITUTO DE INFORMÁTICA UFRGS

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

PRACTICAL ANALYSIS OF EMBEDDED MICROCONTROLLERS AGAINST CLOCK GLITCHING ATTACKS

Ricardo Gomes da Silva

me [at] rgsilva [dot] com
@debugweshell

H2HC, Hackers to Hackers Conference, 11ª edição, 2014

WHO AM I?

Agenda

- Introduction
- Glitcher design
- The setup
- Attacking the target
- Conclusion

INTRODUCTION

Introduction

- Microcontrollers
  - Extremely popular, present nearly everywhere
  - Embedded devices
  - Driven by a clock signal
- Clock signals
  - Digital circuits
  - Synchronization

Clock glitching

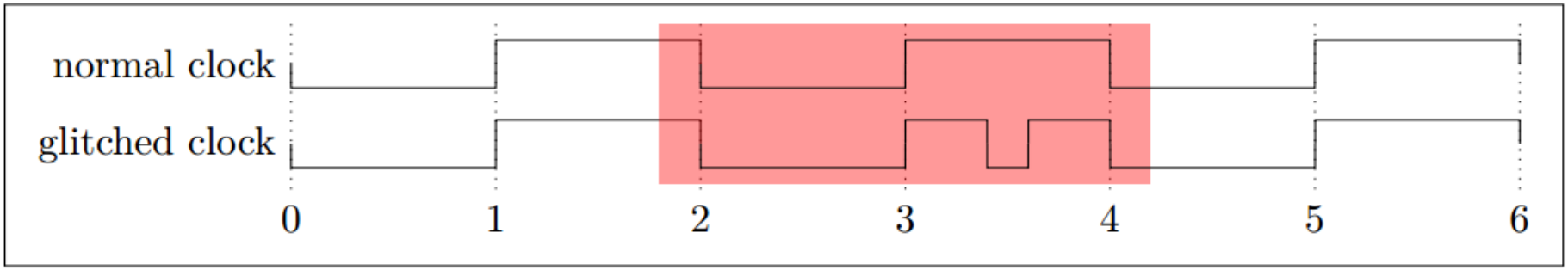
- Non-invasive attack
  - No need for big tools
  - No need for opening the chip
  - It *usually* does not destroy the chip
- Unexpected behavior in the clock signal
  - Changes in the frequency
  - Goal – exceed maximum frequency of the chip (*by far*)

Clock glitching

- Clock is faster than data propagation
  - Instruction or data is corrupted
  - New data does not arrive in time

- Security
  - Target one or multiple instructions – timing-critical attack!
  - Corrupted instructions (unknown/different opcodes)
    - CPU halts or instruction is skipped
    - AVR microcontrollers – unknown opcode is replaced by NOP
  - Control-flow can be altered
    - Bypass security measures
    - Exit or repeat loops

Clock glitching



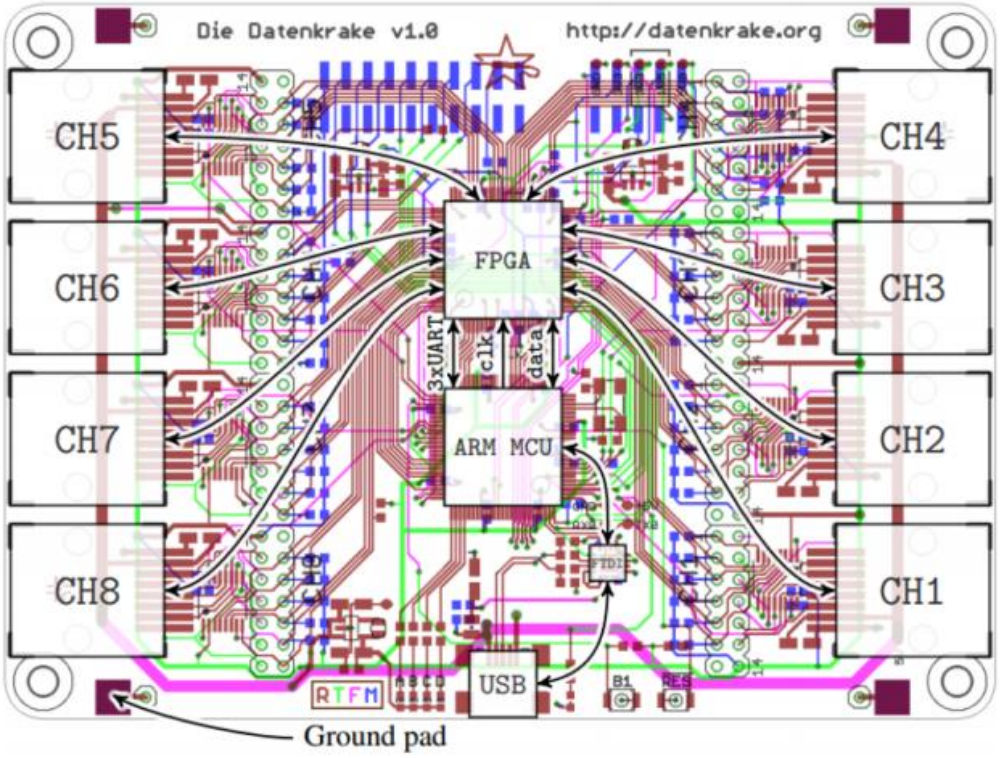
GLITCHER DESIGN

Glitcher design

- Modular design
- Fully configurable glitches
  - *Delay*: when should we glitch?
  - *Width*: for how long?
  - *Mode*: how exactly is the output signal?
- Support to multiple glitches within one execution
  - FIFO to hold glitches in sequence
- Solution for synchronization issue
  - External reset and boot trigger

Glitcher design

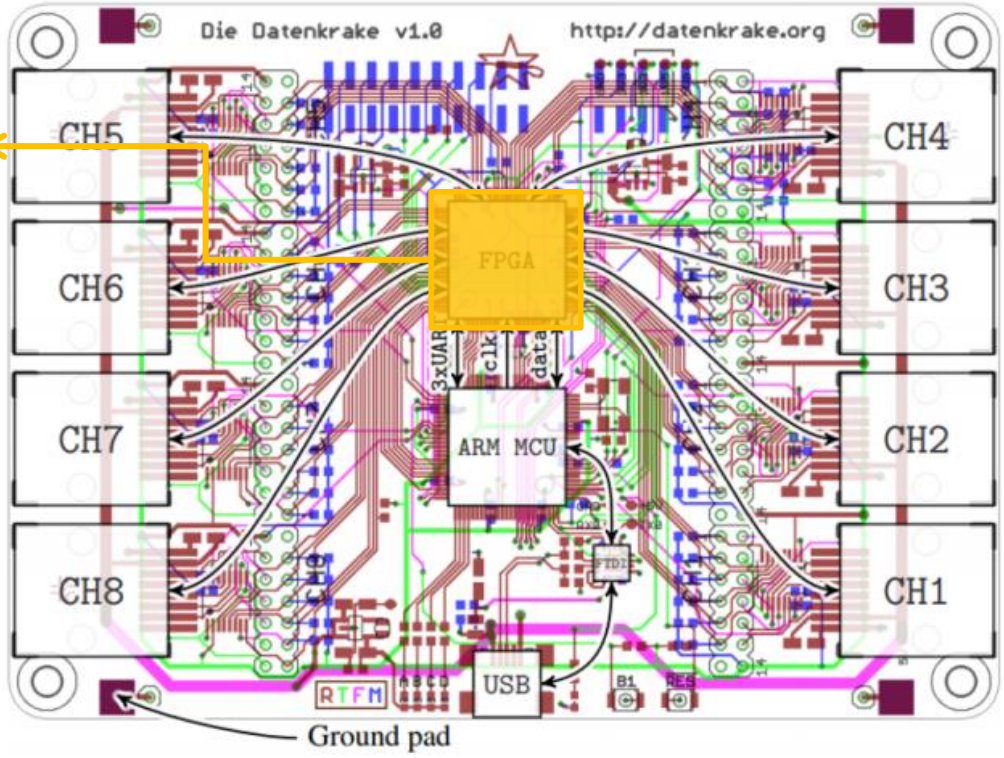
- Development platform: *Die Datenkrake*
  - Open-source hardware and software toolchain
  - Focus in reverse-engineering and security analysis



Glitcher design

- Development platform: *Die Datenkrake*
  - Open-source hardware and software toolchain
  - Focus in reverse-engineering and security analysis

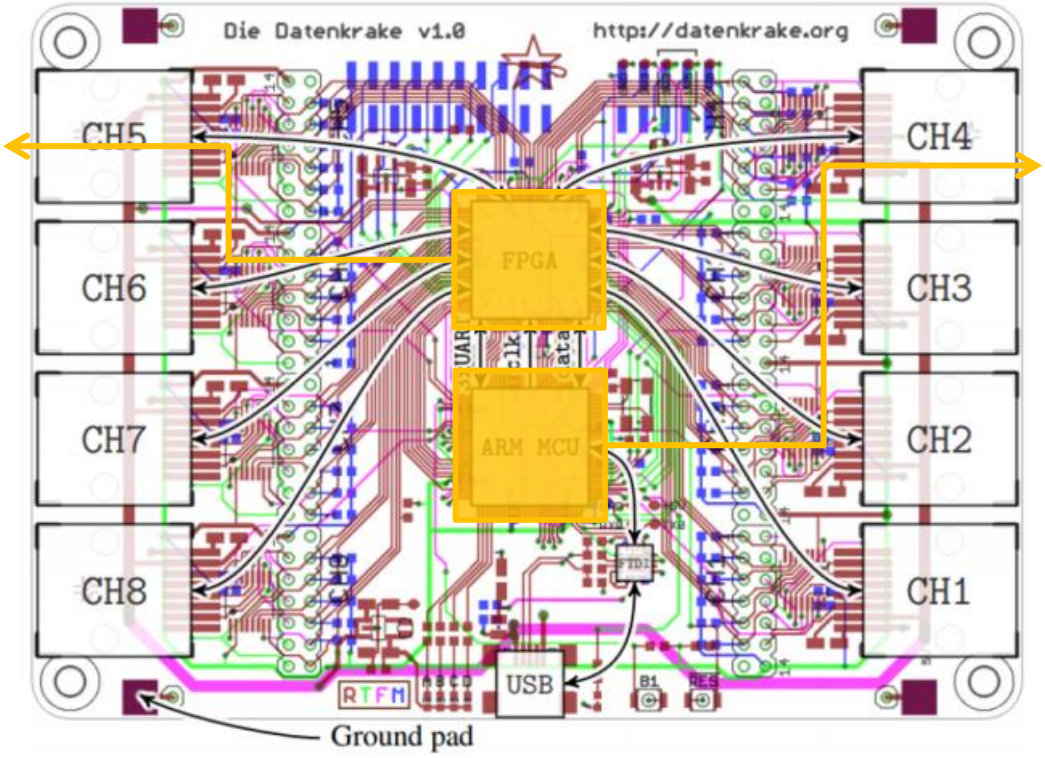
- Timing-critical stuff
- PLL is available
- FIFO generation tools
- Wishbone bus



Glitcher design

- Development platform: *Die Datenkrake*
  - Open-source hardware and software toolchain
  - Focus in reverse-engineering and security analysis

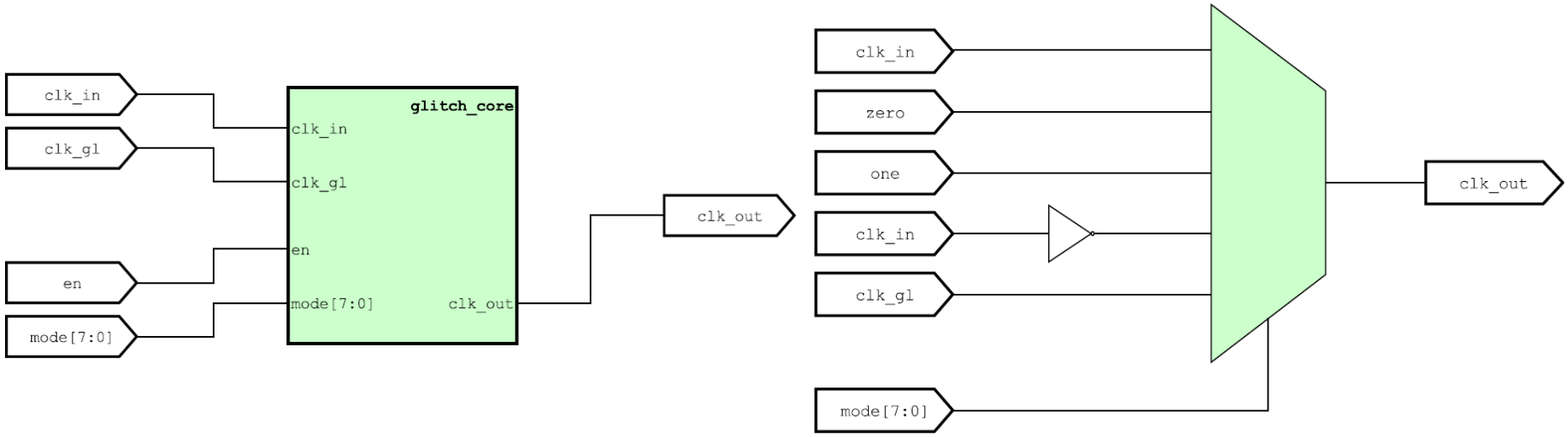
- Timing-critical stuff
- PLL is available
- FIFO generation tools
- Wishbone bus



- Real-time OS
- UART for CLI
- Control and monitor

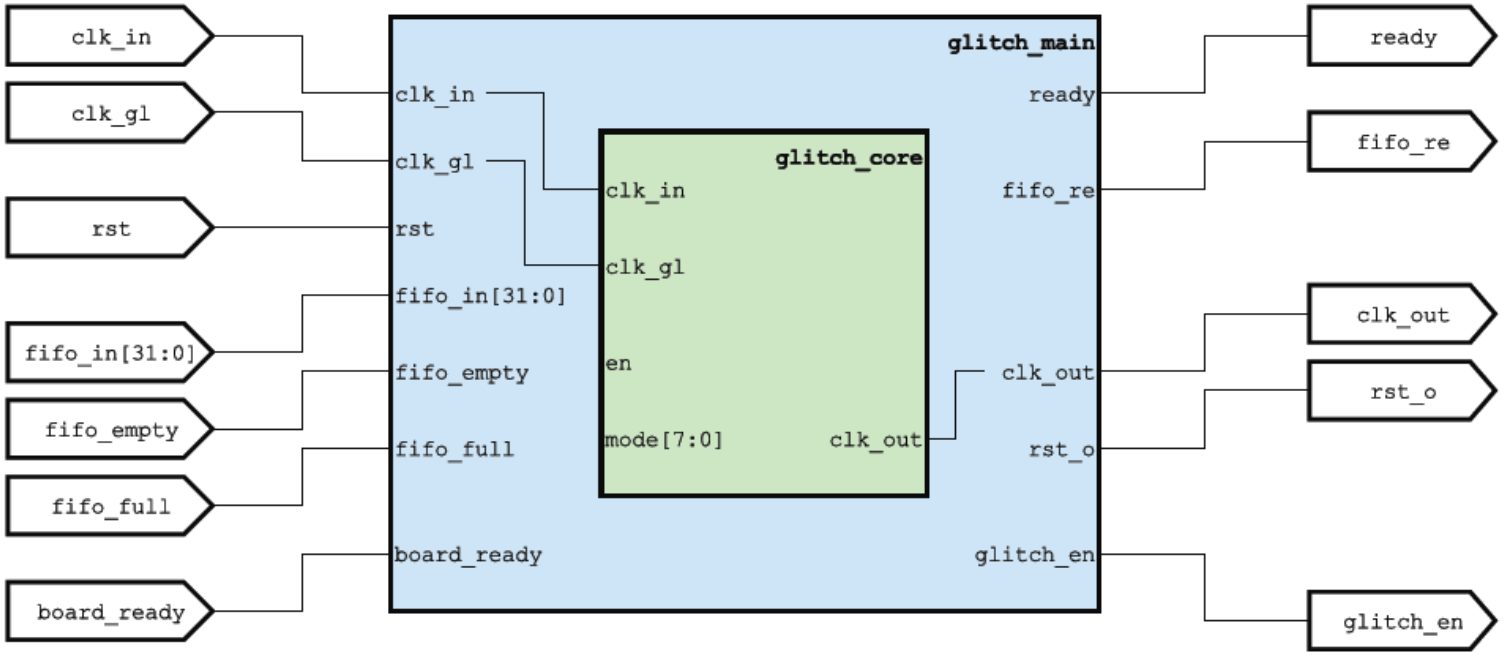
Glitcher design

- Core module
  - Generates the clock signal and the glitches
  - Multiplexer to switch through sources
  - Hard to interface and/or configure



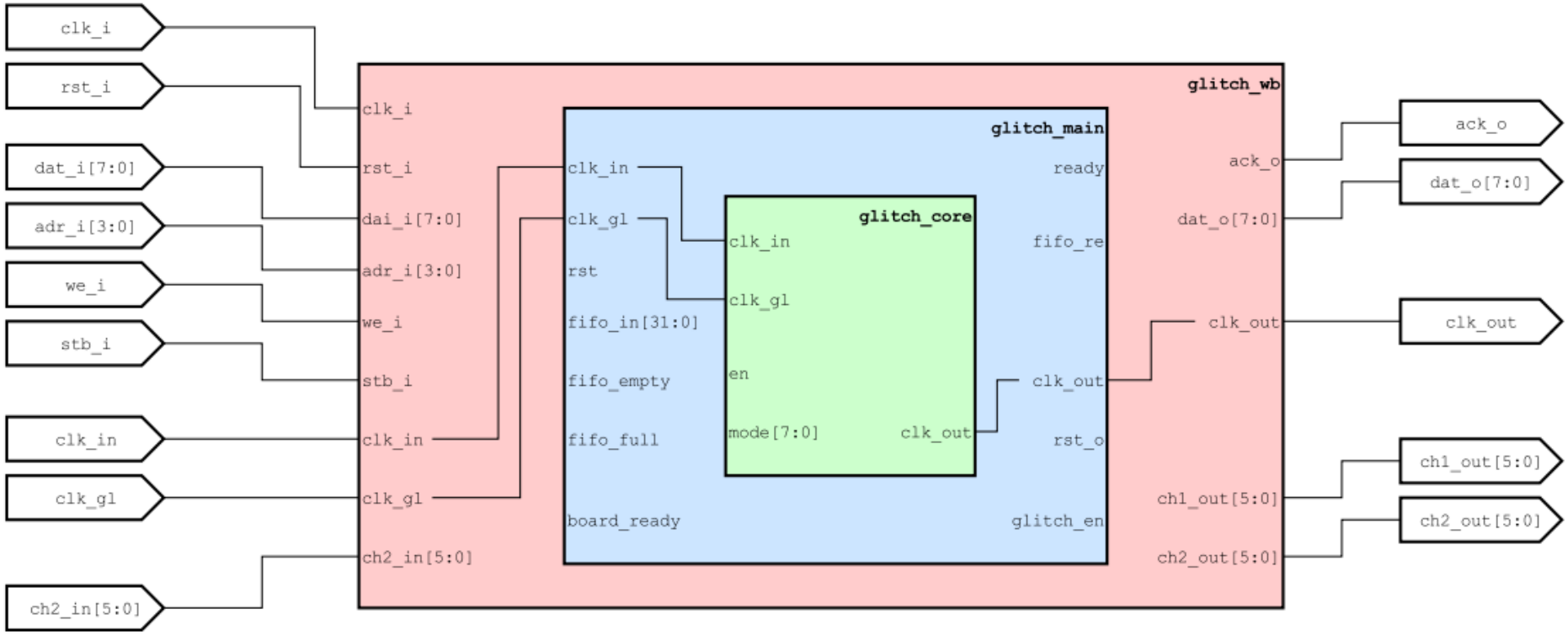
Glitcher design

- Main module
  - Concept of *delay*, *width* and *mode* as glitch configuration
  - FIFO for multiple glitches
  - External target reset and trigger signals



Glitcher design

- Wishbone Wrapper
  - Interface for the DDK



THE SETUP

The setup – target

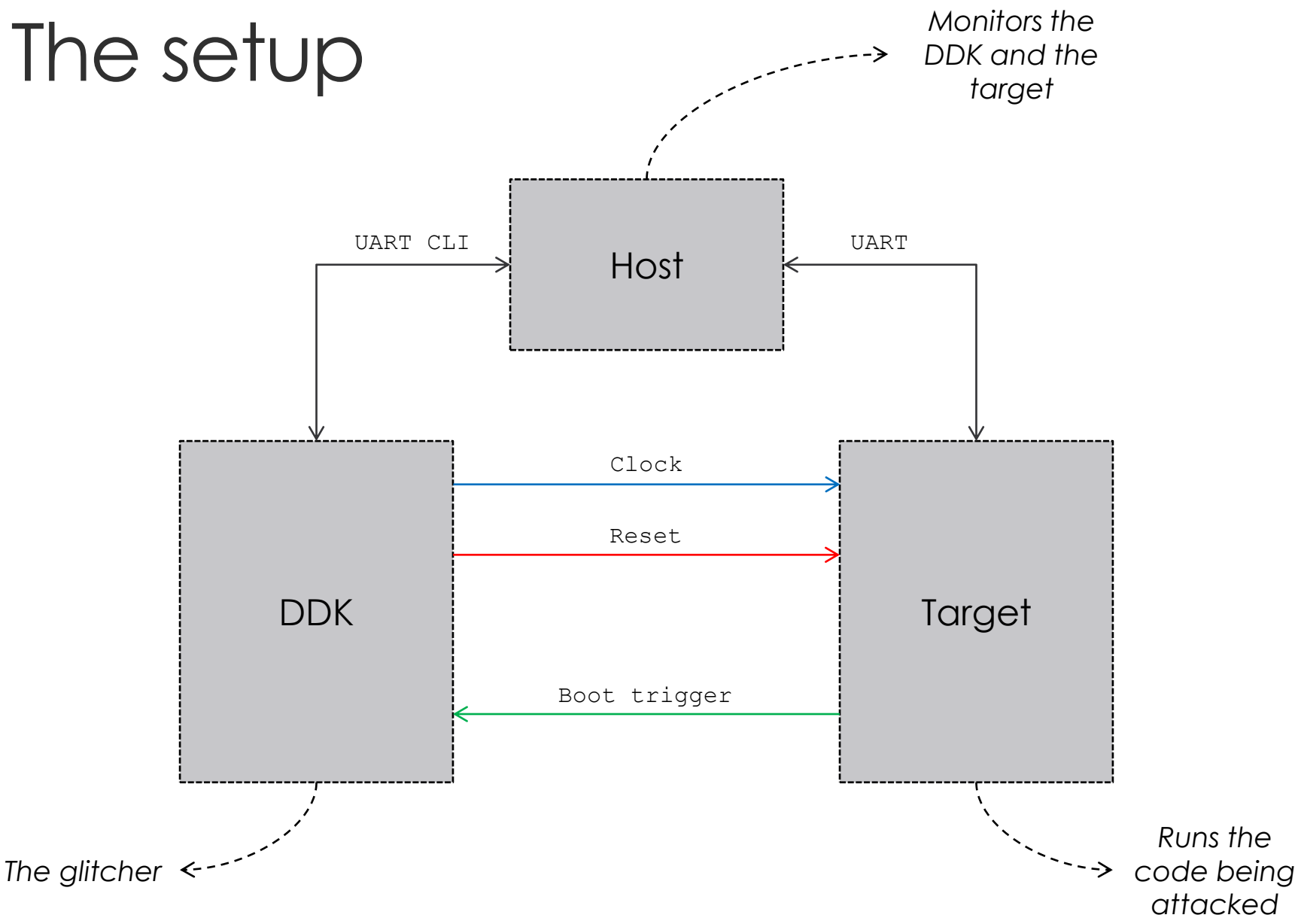
- Atmel XMEGA-A1 Xplained
  - ATxmega128A1 microcontroller evaluation kit
- Runs the code to be attacked
  - Previously known (although not really necessary)
  - No protections (real-time platform)
- Uses the glitcher as clock source
  - 33 MHz for normal execution, 99 MHz for glitching
- Can be externally reset
- Can be externally monitored
  - Boot trigger
  - GPIO or UART for glitch detection

The setup – host

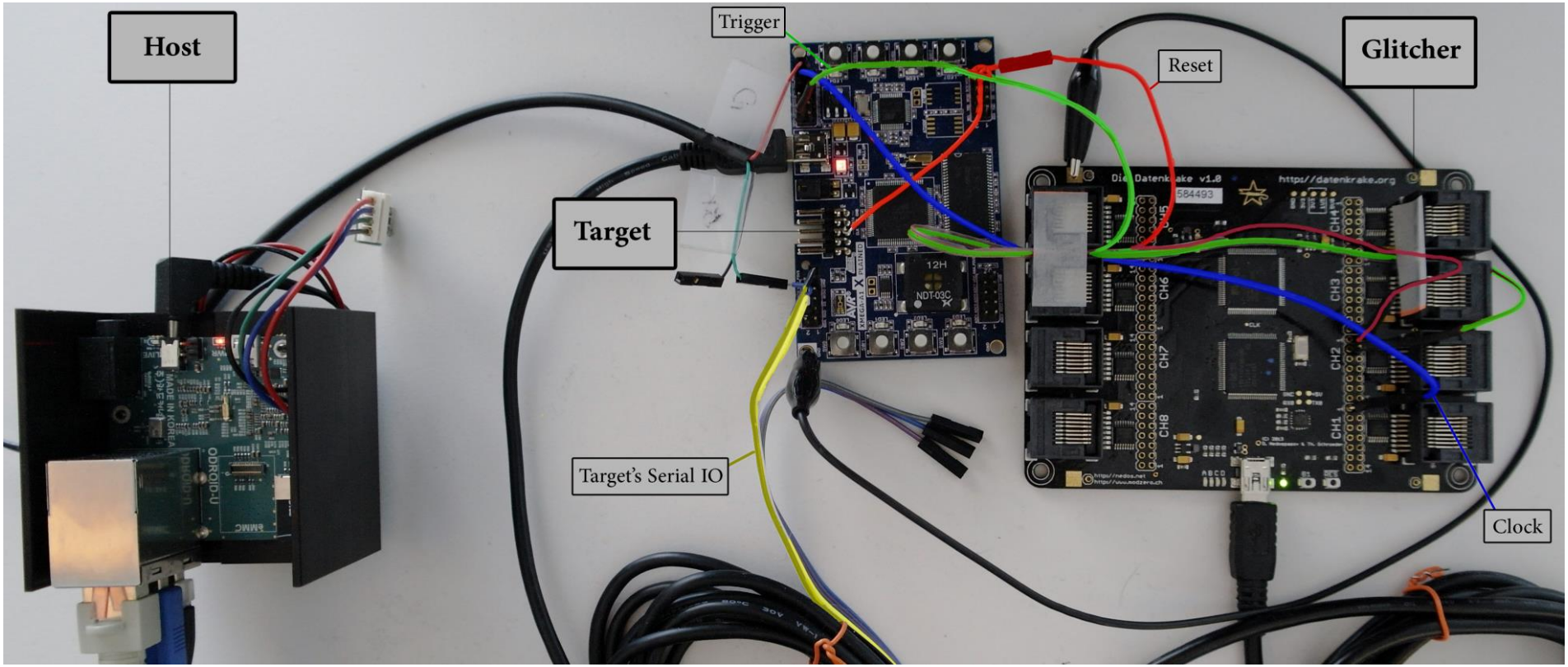
- Exact glitch position is unknown
  - Multiple scenarios are valid
    - Pipeline – fetch, decode, execute
  - Thousands of combinations must be tested
    - Brute-force attacks within a range

- A script is necessary
  - Generate all the combinations within a range
  - Try each one of them with N repetitions
  - Log the result from the target (GPIO or Serial)
  - Periodically executes a self-test
    - Protection against instabilities

The setup


Monitors the DDK and the target
Host
UART CLI
UART
DDK
Target
Clock
Reset
Boot trigger
The glitcher
Runs the code being attacked

The setup


Host
Trigger
Reset
Glitcher
Target
Target's Serial IO
Clock

ATTACKING THE TARGET

Attacking the target

- I – Baby steps (handmade AVR assembly)
- II – Proof of concept (handmade C code)
- III – Real world (3rd-party C code)

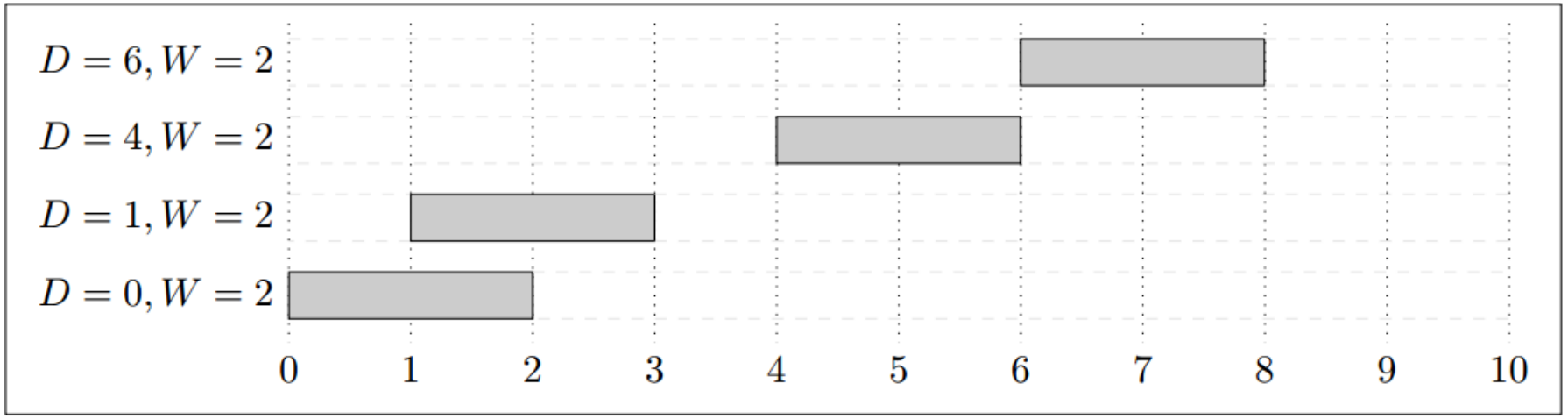
Baby steps – JMP (1)

```
1  breakme:
2      jmp breakme

4  glitched:
5      ldi r24, 0b11111100
6      sts PORTD_OUTSET, r24
7      rjmp glitched
```

Infinite loop

Should never be executed!

Baby steps – JMP (1)



- Multiple combinations are possible
- A pattern is visible
  - Infinite loop makes it easy to hit the instruction

Baby steps – JMP (2)

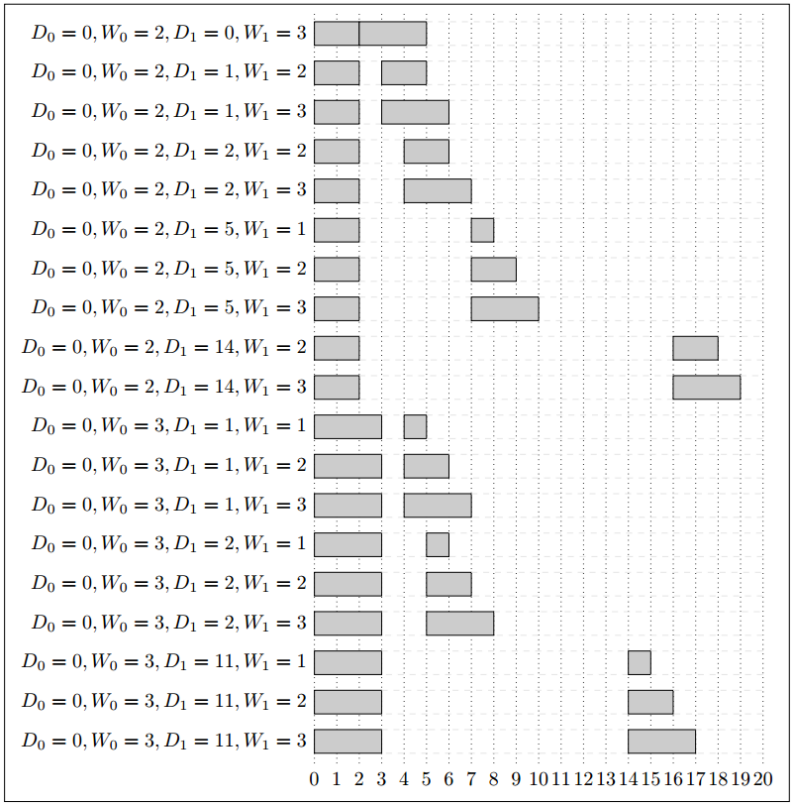
```
1   breakme:
2       jmp breakme

4   second:
5       ldi r24, 0x00
6       sts PORTE_OUTSET, r24
7       ldi r24, 0xFF
8       sts PORTE_OUTSET, r24

10      jmp second

12  glitched:
13      ldi r24, 0b11111100
14      sts PORTD_OUTSET, r24
15      rjmp glitched
```

Infinite loop

Second infinite loop (with LEDs)

Should never be executed!

Baby steps – JMP (2)



- Two patterns visible
  - Gaps represent the current iteration

Proof of concept – strcpy

```
1  char secret0[] = "000000";
2  char secret1[] = "111111";
3  char secret2[] = "222222";
4  char foobar[]  = "foobar";
5  char secret3[] = "333333";
6  char secret4[] = "444444";
7  char secret5[] = "555555";

9  int main()
10 {
11     const char buffer[256] = {0};

13     strcpy(buffer, foobar);
14     fputs(buffer, stdout);

16     while (1) { asm volatile("nop\n"); }
17 }
```

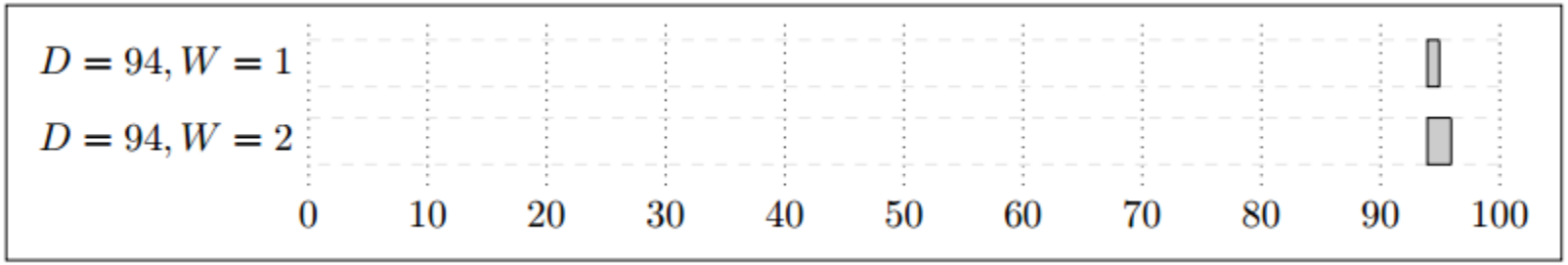
Proof of concept – strcpy

```
1  loop:
2      ld r0, Z+
3      st X+, r0
4      and r0, r0
6      brne loop
```

1. Load the next byte and increase the source pointer
2. Store the byte and increase the destination pointer
3. Check if it's zero – if not, continue the loop

- What happens if we glitch the `LD` or the `AND` instructions?
- Pipeline vs. `NOP` padding

Proof of concept – strcpy



- Not so many combinations to glitch it
  - One specific instruction is being glitched
    - Either `LD` or `AND`
  - We must wait until he hit the null-byte
    - The bigger the string, the more time we need to wait

Proof of concept – strcpy

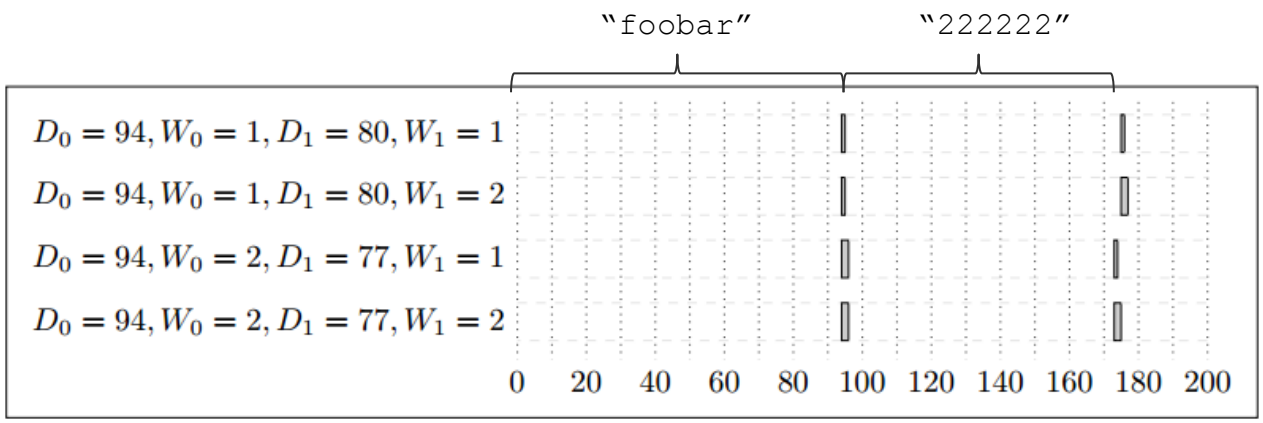
```
1  + Running test 189 of 1502
2  ? Got something from target, len = 13 [66 6F 6F 62 61 72
3    10 32 32 32 32 32 32] [foobar 222222]
4  + Accuracy: 100.0%

6  + Running test 190 of 1502
7  ? Got something from target, len = 13 [66 6F 6F 62 61 72
8    10 32 32 32 32 32 32] [foobar 222222]
9  + Accuracy: 100.0%
```

- Success on extracting a "secret" string: `222222`
- The null-byte is now `0x10`
  - Strong indicator for a `LD` glitch and not an `AND` instruction
- Can we repeat it? If yes, how far can we go?

```
char secret0[] = "000000";
char secret1[] = "111111";
char secret2[] = "222222";
char foobar[]  = "foobar";
char secret3[] = "333333";
char secret4[] = "444444";
char secret5[] = "555555";
```

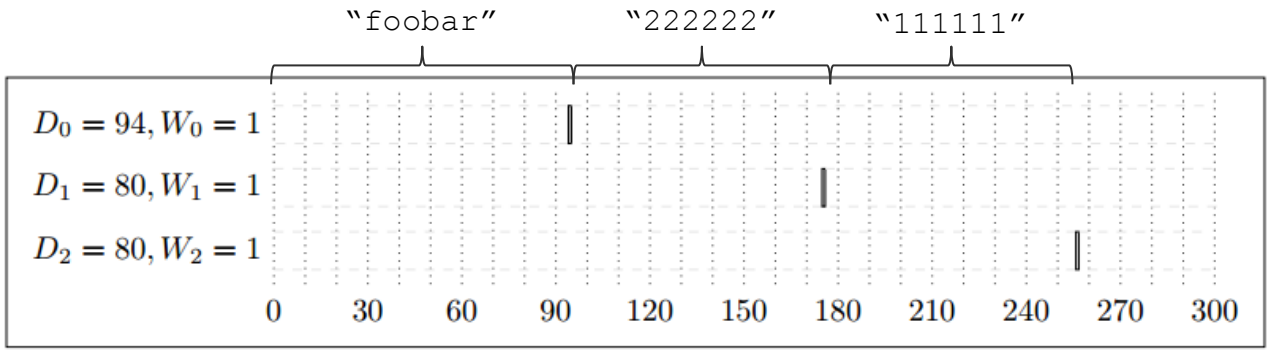
Proof of concept – strcpy

```
1  + Running test 1777 of 4444
2  ? Got something from target, len = 20 [66 6F 6F 62 61 72
3    10 32 32 32 32 32 32 10 31 31 31 31 31 31] [foobar 222
4    222 111111]
5  + Accuracy: 100.0%

7  + Running test 1778 of 4444
8  ? Got something from target, len = 20 [66 6F 6F 62 61 72
9    10 32 32 32 32 32 32 10 31 31 31 31 31 31] [foobar 222
10   222 111111]
11 + Accuracy: 100.0%
```



Proof of concept – strcpy

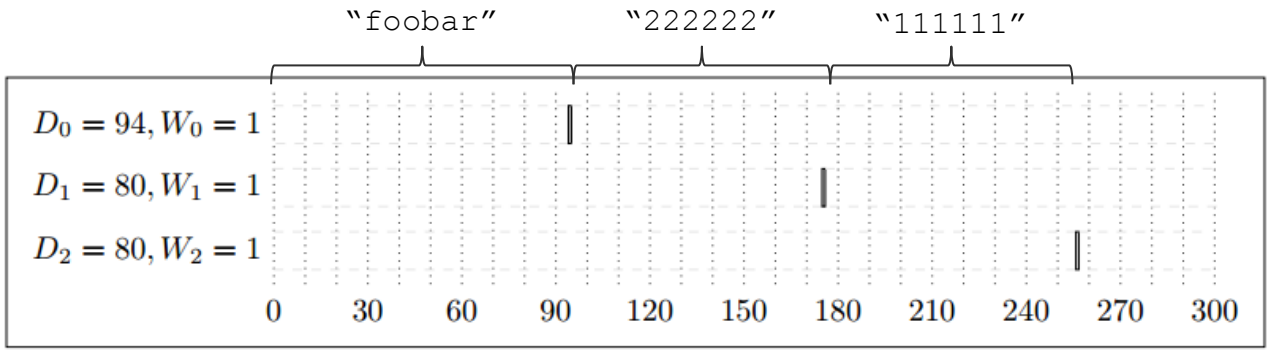
```
1  + Running test 81 of 101
2
3  ? Got something from target, len = 25 [66 6F 6F 62 61 72
4    50 32 32 32 32 32 32 31 31 31 31 31 31 30 30 30 30 30
5    30] [foobar 222222111111000000]
6
7  ? Got something from target, len = 27 [66 6F 6F 62 61 72
8    50 32 32 32 32 32 32 10 31 31 31 31 31 31 10 30 30 30
9    30 30 30] [foobarP222222 111111 000000]
10
11 ? Got something from target, len = 25 [66 6F 6F 62 61 72
12   50 32 32 32 32 32 32 31 31 31 31 31 31 30 30 30 30 30
13   30] [foobarP222222111111000000]
14
15 ? Got something from target, len = 26 [66 6F 6F 62 61 72
16   32 32 32 32 32 32 10 31 31 31 31 31 31 10 30 30 30 30
17   30 30] [foobar222222 111111 000000]
18
19 + Accuracy: 100.0%
```



Proof of concept – strcpy

Byte `0x50` instead of `0x10` as separator

```
1  + Running test 81 of 101

3  ? Got something from target, len = 25 [66 6F 6F 62 61 72
4    50 32 32 32 32 32 32 31 31 31 31 31 31 30 30 30 30 30
5    30] [foobar 222222111111000000]

7  ? Got something from target, len = 27 [66 6F 6F 62 61 72
8    50 32 32 32 32 32 32 10 31 31 31 31 31 31 10 30 30 30
9    30 30 30] [foobarP222222 111111 000000]

11 ? Got something from target, len = 25 [66 6F 6F 62 61 72
12   50 32 32 32 32 32 32 31 31 31 31 31 31 30 30 30 30 30
13   30] [foobarP222222111111000000]

15 ? Got something from target, len = 26 [66 6F 6F 62 61 72
16   32 32 32 32 32 32 10 31 31 31 31 31 31 10 30 30 30 30
17   30 30] [foobar222222 111111 000000]

19 + Accuracy: 100.0%
```

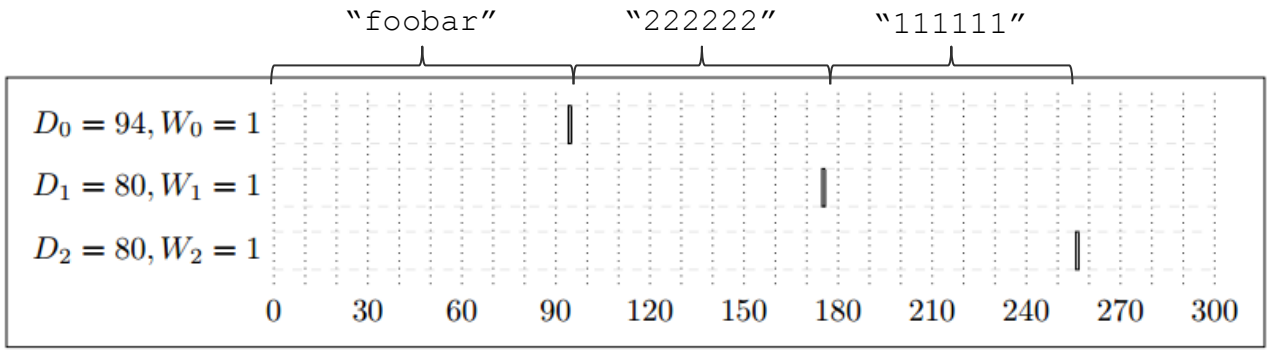


Proof of concept – strcpy

```
1  + Running test 81 of 101
2
3  ? Got something from target, len = 25 [66 6F 6F 62 61 72
4    50 32 32 32 32 32 32 31 31 31 31 31 31 30 30 30 30 30
5    30] [foobar 222222111111000000]
6
7  ? Got something from target, len = 27 [66 6F 6F 62 61 72
8    50 32 32 32 32 32 32 10 31 31 31 31 31 31 10 30 30 30
9    30 30 30] [foobarP222222 111111 000000]
10
11 ? Got something from target, len = 25 [66 6F 6F 62 61 72
12   50 32 32 32 32 32 32 31 31 31 31 31 31 30 30 30 30 30
13   30] [foobarP222222111111000000]
14
15 ? Got something from target, len = 26 [66 6F 6F 62 61 72
16   32 32 32 32 32 32 10 31 31 31 31 31 31 10 30 30 30 30
17   30 30] [foobar222222 111111 000000]
18
19 + Accuracy: 100.0%
```

Byte `0x50` instead of `0x10` as separator

Different results with different lengths



Real world – crypto libraries

- *Practical Second-Order Fault Attack against a Real-World Pairing Implementation (FDTC 2014)*

  Johannes Blömer, Ricardo Gomes da Silva, Peter Günther, Juliane Krämer, Jean-Pierre Seifert

- "First practical fault attack against a complete pairing computation"

- Pairing-based crypto attacked through hardware

- Same setup
  - Extra features on the DDK for profiling
  - Extra size for delay: 32 instead of 8 bits

Real world – crypto libraries

```
/* ... */
call fb4_mul_dxs
.LVL43:
/* decrement loop counter LSB, MSB */
subi r16,1
sbc r17,__zero_reg__
.loc 1 247 0 discriminator 2
/* jump out of the loop */
breq .+2
/* jump loop begin */
rjmp .L2
.LEB2:
.loc 1 486 0
/* clean stack */
subi r28,36
sbci r29,-2
out __SP_L__,r28
out __SP_H__,r29
pop r29
/* ... */
/* exponentiation call */
call etat_exp
```

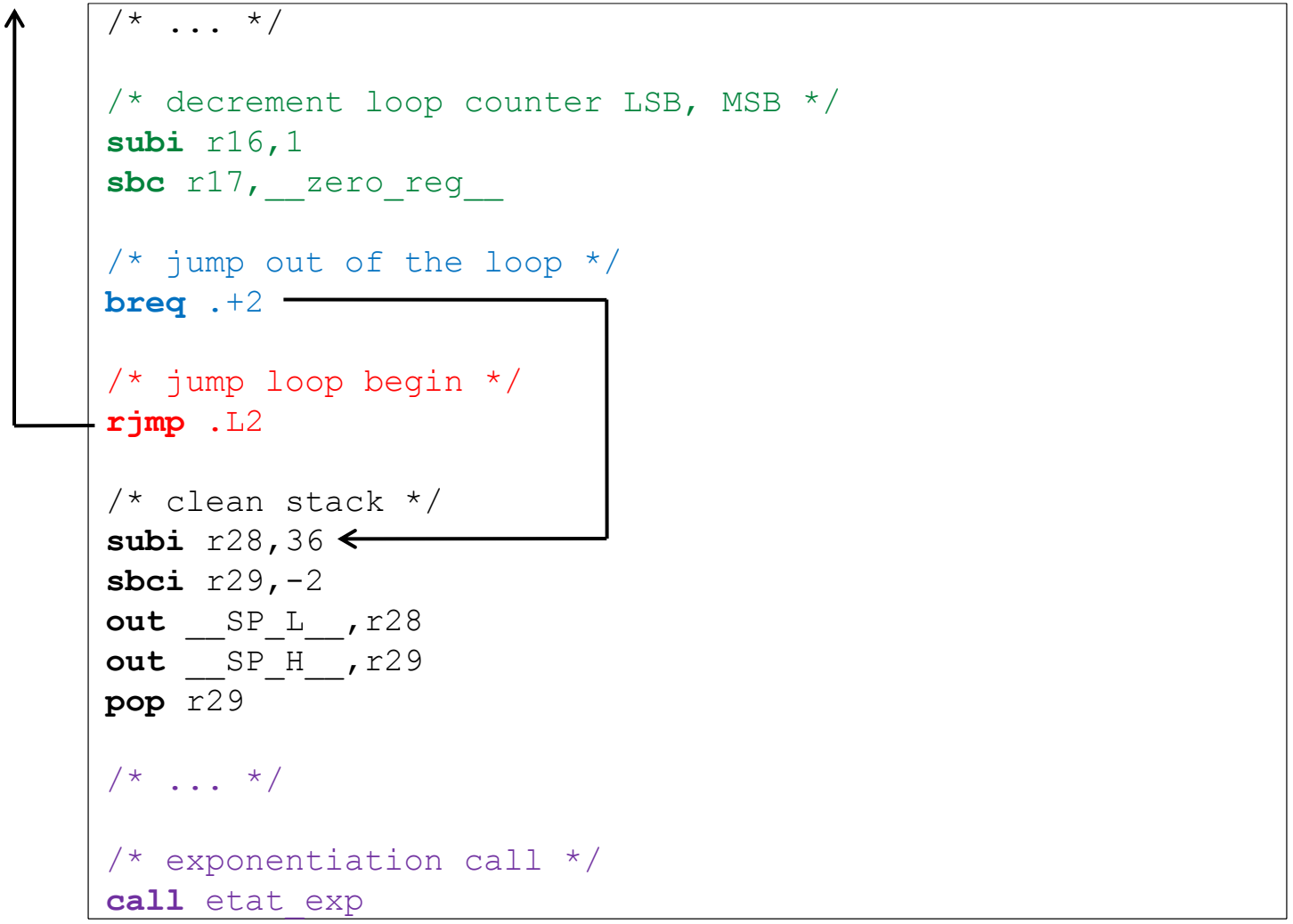
Real world – crypto libraries

```
/* ... */

/* decrement loop counter LSB, MSB */
subi r16,1
sbc r17,__zero_reg__

/* jump out of the loop */
breq .+2

/* jump loop begin */
rjmp .L2

/* clean stack */
subi r28,36
sbci r29,-2
out __SP_L__,r28
out __SP_H__,r29
pop r29

/* ... */

/* exponentiation call */
call etat_exp
```

Real world – crypto libraries

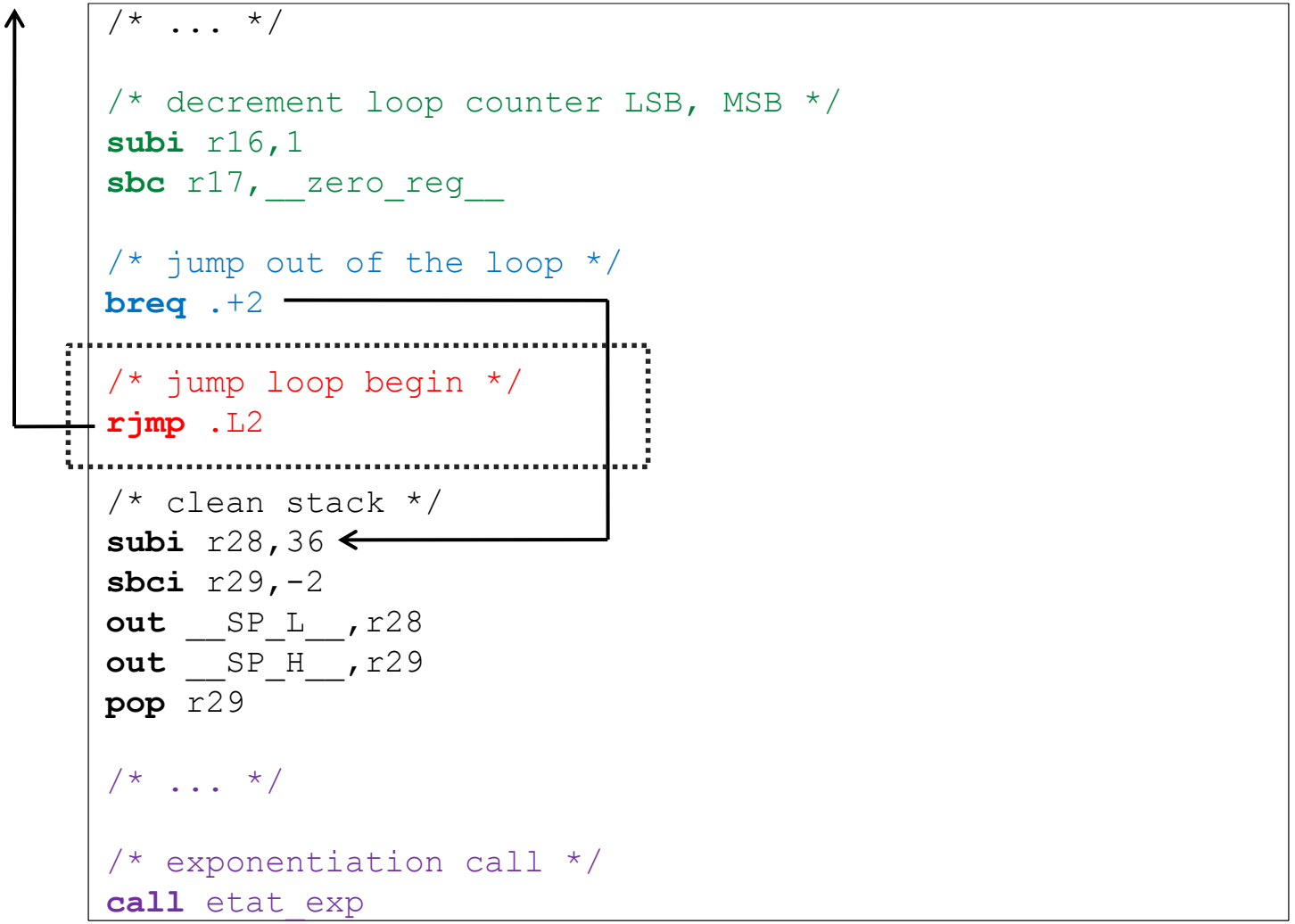
```
/* ... */

/* decrement loop counter LSB, MSB */
subi r16,1
sbc r17,__zero_reg__

/* jump out of the loop */
breq .+2

/* jump loop begin */
rjmp .L2

/* clean stack */
subi r28,36
sbci r29,-2
out __SP_L__,r28
out __SP_H__,r29
pop r29

/* ... */

/* exponentiation call */
call etat_exp
```

Real world – crypto libraries

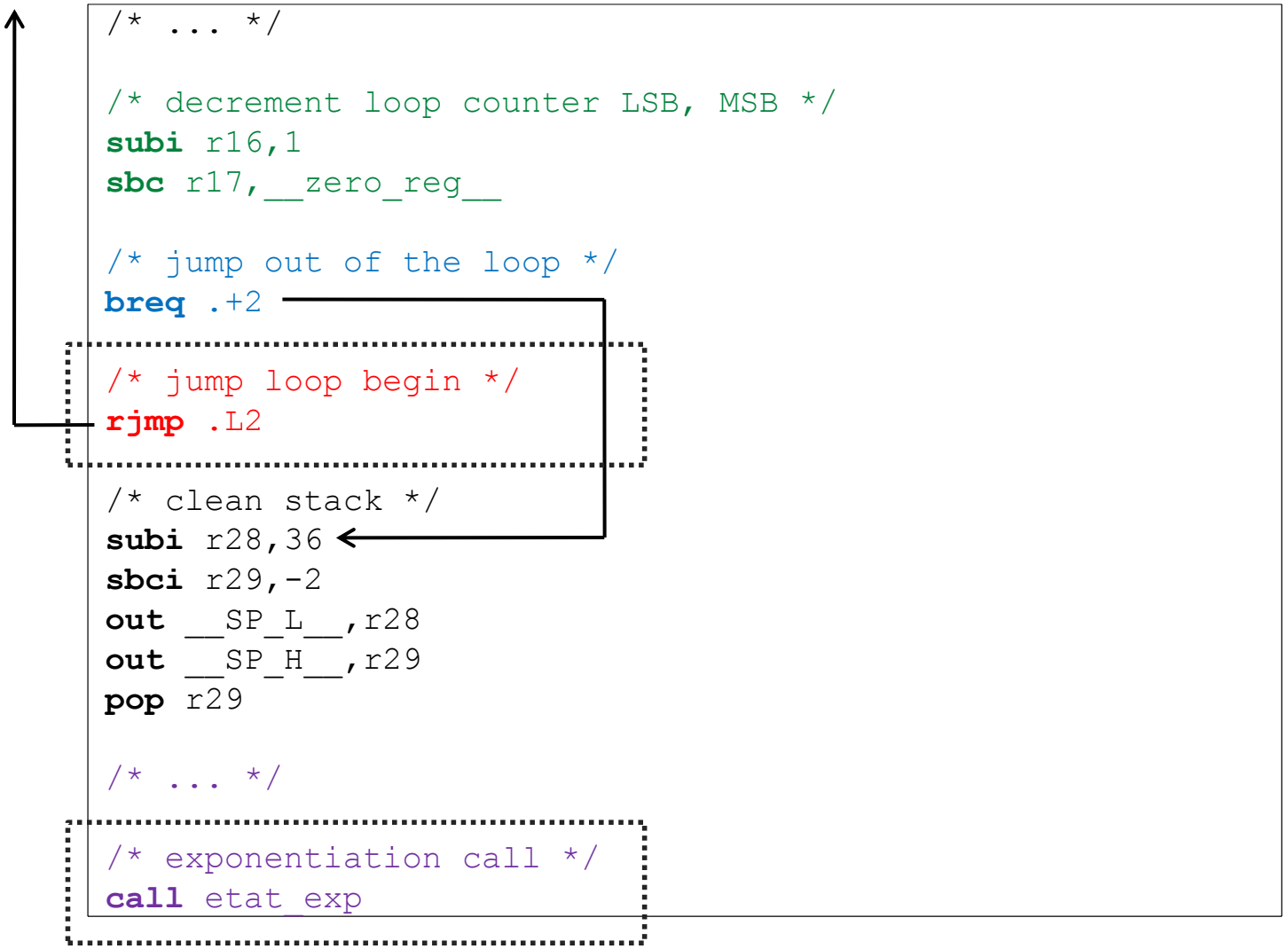


```
/* ... */

/* decrement loop counter LSB, MSB */
subi r16,1
sbc r17,__zero_reg__

/* jump out of the loop */
breq .+2

/* jump loop begin */
rjmp .L2

/* clean stack */
subi r28,36
sbci r29,-2
out __SP_L__,r28
out __SP_H__,r29
pop r29

/* ... */

/* exponentiation call */
call etat_exp
```

Real world – crypto libraries



Real world – crypto libraries



CONCLUSION

Conclusion

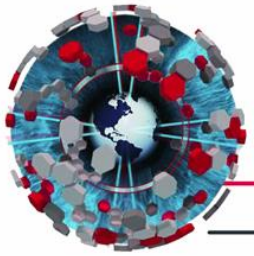
- A clock glitching platform was fully developed
  - Both hardware and software are open-source
  - Support to multiple glitches within the same execution
  - Automatic attacking and monitoring
    - Log of attacks and basic analysis of success/failure
    - Automatic self-testing

- We successfully attacked an AVR microcontroller
  - Attacks are repeatable and stable
  - Real world targets can also be attacked in this setup
  - Exiting loops and dumping memory are just examples

Conclusion

- Why is this a good solution?
  - Cheap solution
  - No need for big tools
  - Non-destructive attack
  - No need to open or probe inside the chip
  - Did I say it's open-source? :-)

```
Glitcher ARM:    https://github.com/rgsilva/ddk-arm/
Glitcher FPGA:   https://github.com/rgsilva/ddk-fpga/
DDK source:      http://datenkrake.org/
```



11ª EDIÇÃO 2014

H2HC

HACKERS TO HACKERS CONFERENCE

.inf INSTITUTO DE INFORMÁTICA UFRGS

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

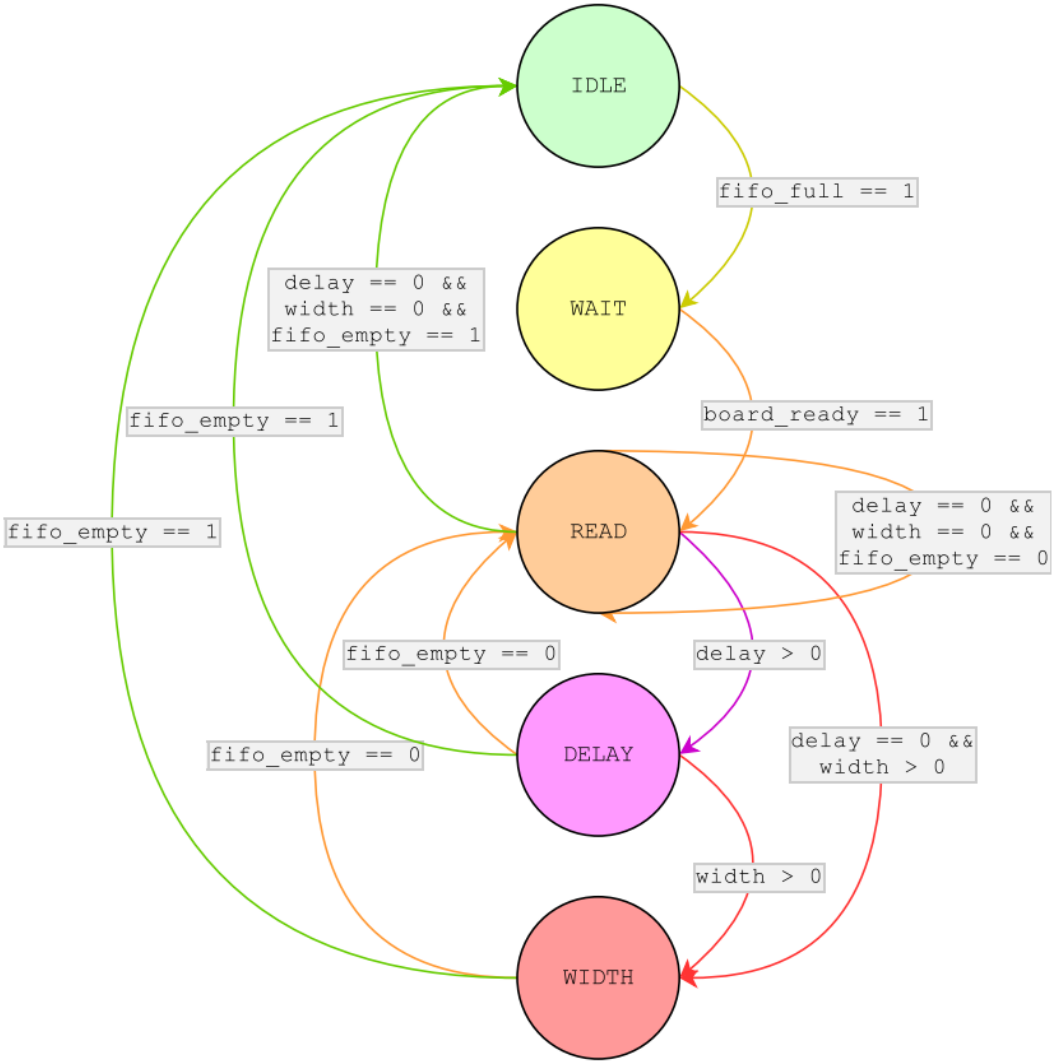
PRACTICAL ANALYSIS OF EMBEDDED MICROCONTROLLERS AGAINST CLOCK GLITCHING ATTACKS

Ricardo Gomes da Silva

me [at] rgsilva [dot] com
@debugweshell

H2HC, Hackers to Hackers Conference, 11ª edição, 2014

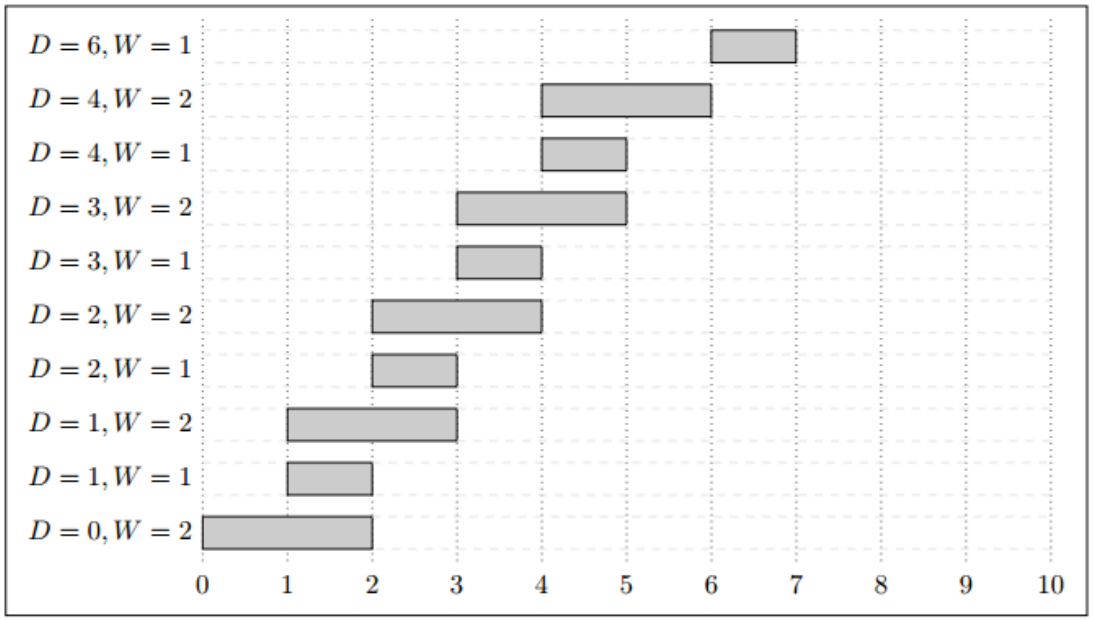
Extra – state machine



Extra – BRNE

```
1 breakme:
2     cpi r24, 0xFF
3     breq breakme
```

```
1 volatile unsigned int i = 255;
2
3 while (i == 255) {
4     // Do something.
5 }
```



Extra – padded strcpy

```
1  char* strcpy(char *dest, const char *source)
2  {
3      asm("nop\n"); asm("nop\n");
4      asm("movw r30, r22\n");
5      asm("nop\n"); asm("nop\n");
6      asm("movw r26, r24\n");
7      asm("nop\n"); asm("nop\n");

9      asm("strcpy_loop:\n");
10     asm("ld r0, Z+\n");
11     asm("nop\n"); asm("nop\n");
12     asm("st X+, r0\n");
13     asm("nop\n"); asm("nop\n");
14     asm("and r0, r0\n");
15     asm("nop\n"); asm("nop\n");
16     asm("brne strcpy_loop\n");
17     asm("nop\n"); asm("nop\n");
18 }
```

Extra – New modes