

# A Practical Second-Order Fault Attack against a Real-World Pairing Implementation

Peter Günther

joint work with

Johannes Blömer

Ricardo Gomes da Silva

Juliane Krämer

Jean-Pierre Seifert

University of Paderborn

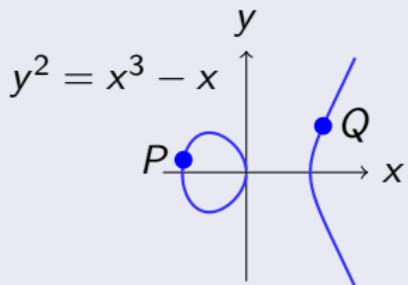
Technical University of Berlin

FDTc 2014, September 23, 2014, Busan

# The role of the final exponentiation

## Two step pairing computation

$$\begin{aligned} e : \mathcal{E}(\mathbb{F}_{p^k}) \times \mathcal{E}(\mathbb{F}_{p^k}) &\rightarrow \mathbb{F}_{p^k}^* \\ (P, Q) &\mapsto f_{r,P}(Q)^d \end{aligned}$$



- ①  $\alpha \leftarrow f_{r,P}(Q)$  (Miller step)
  - ②  $\beta \leftarrow \alpha^d$  with  $d = (p^k - 1)/r$  (final exponentiation)
- 
- ① Computes non-degenerate, bilinear mapping to  $\mathbb{F}_{p^k}^*/(\mathbb{F}_{p^k}^*)^r$ .
  - ② Maps equivalence classes  $\mathbb{F}_{p^k}^*/(\mathbb{F}_{p^k}^*)^r$  to unique representatives in  $\mu_r$ .

# Fault attacks on pairings

## Cryptanalysis of pairings

Inversion of both steps is required

- ① Search secret  $Q$  as one solution of  $f_{r,P}(x,y) = \alpha$   
Problem: Function  $f_{r,P}(x,y)$  has huge degree  $r$
  - ② Inversion of final exponentiation  $(\cdot)^d = \beta$   
Problem: Difficult to identify the correct  $d$ -th root  $\alpha$
- ⇒ 2nd order attacks required

## Cryptanalysis of pairings with fault attacks

- ① Reduce degree of  $f_{r,P}(x,y)$  by modification of  $r, P, Q$
- ② Make  $(\cdot)^d$  as injective as possible

# Fault attacks on pairings

## Cryptanalysis of pairings

Inversion of both steps is required

- ① Search secret  $Q$  as one solution of  $f_{r,P}(x, y) = \alpha$   
Problem: Function  $f_{r,P}(x, y)$  has huge degree  $r$
- ② Inversion of final exponentiation  $(\cdot)^d = \beta$   
Problem: Difficult to identify the correct  $d$ -th root  $\alpha$   
 $\Rightarrow$  2nd order attacks required

## Our 2nd order attack

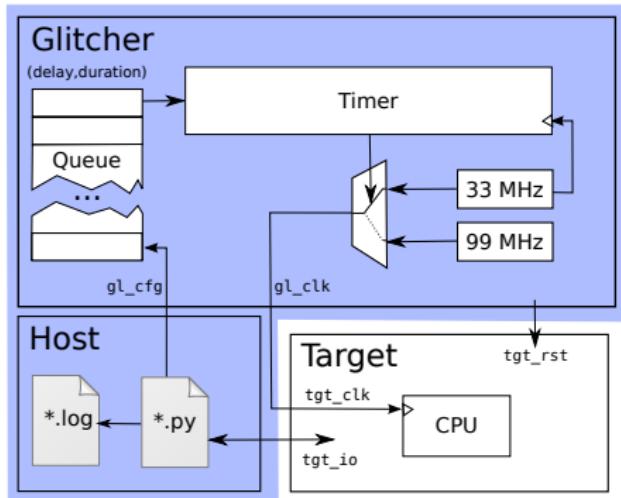
- ① Round reduction of Miller loop: obtain  $f_{r',P}(x, y)$  of degree  $r' = 5$ .
- ② Skipping final exponentiation

# Fault attacks on pairings

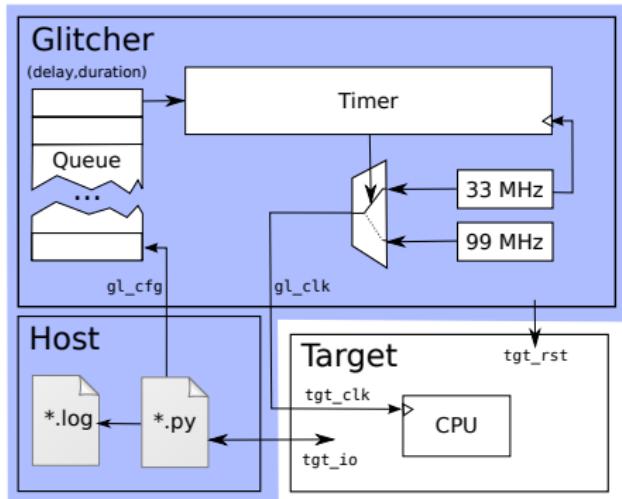
## Outline of our attack

- ① Assumption: attacker with physical access to target (especially CPU clock)
- ② Trigger computation of  $e(P, Q)$  on public argument (e.g.  $P$ ) and secret argument (e.g.  $Q$ )
- ③ Distort computation of  $e(P, Q)$  by clock glitch to obtain  $\beta'$
- ④ Compute secret  $Q$  from  $\beta'$

# Our attack: Schematic Hardware Setup

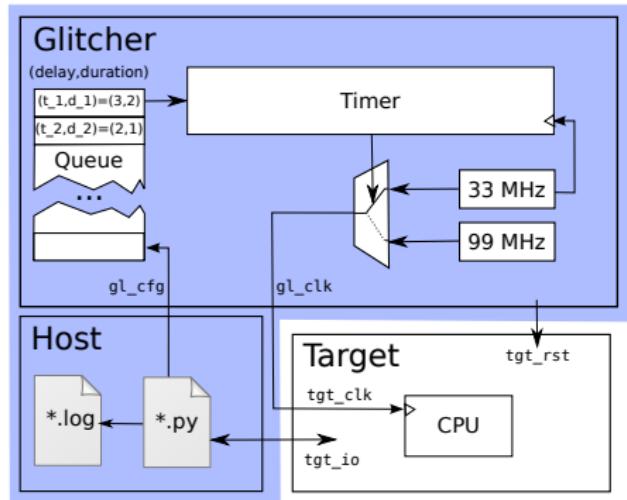


# Our attack: Schematic Hardware Setup

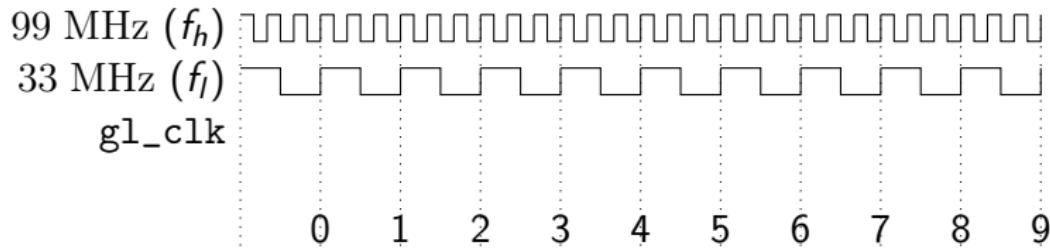


- Mechanism: CPU clock glitching
- Effect: Instruction skips

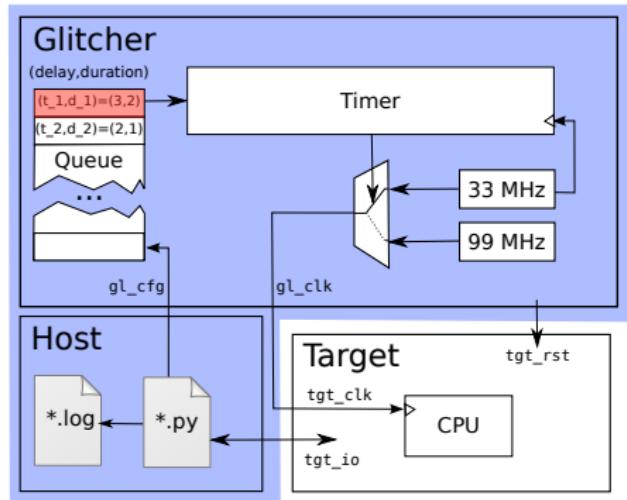
# Our attack: Schematic Hardware Setup



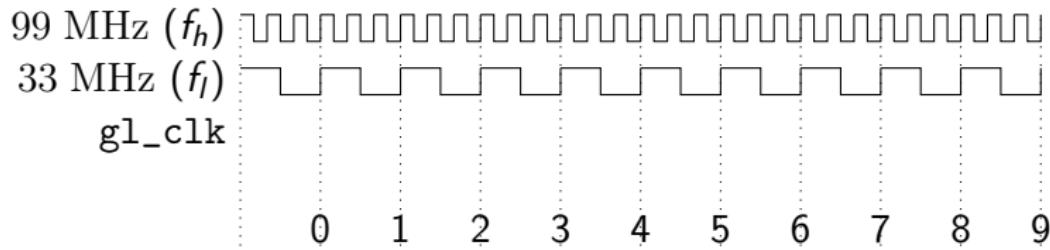
- Mechanism: CPU clock glitching
- Effect: Instruction skips



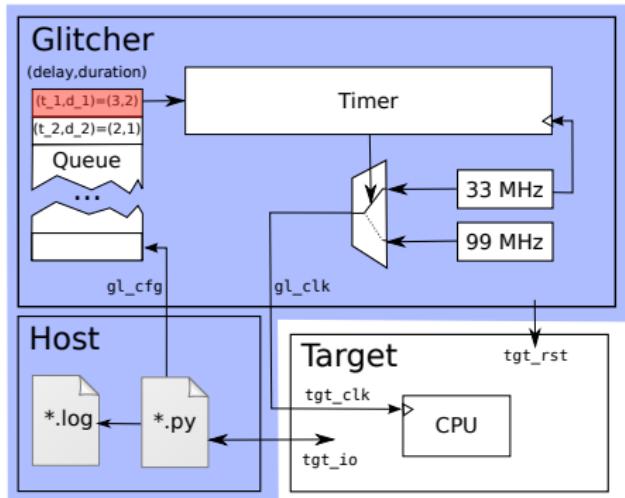
# Our attack: Schematic Hardware Setup



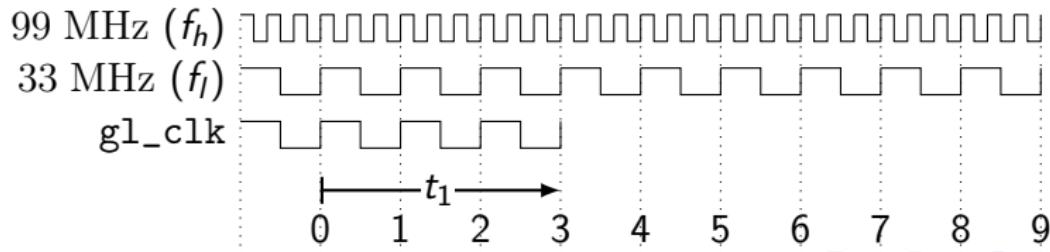
- Mechanism: CPU clock glitching
- Effect: Instruction skips



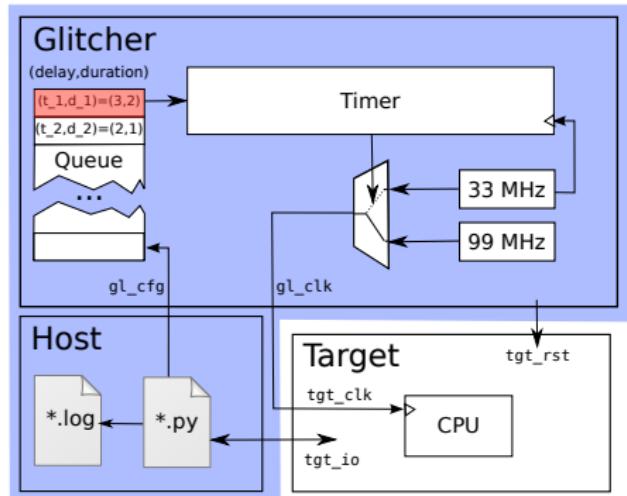
# Our attack: Schematic Hardware Setup



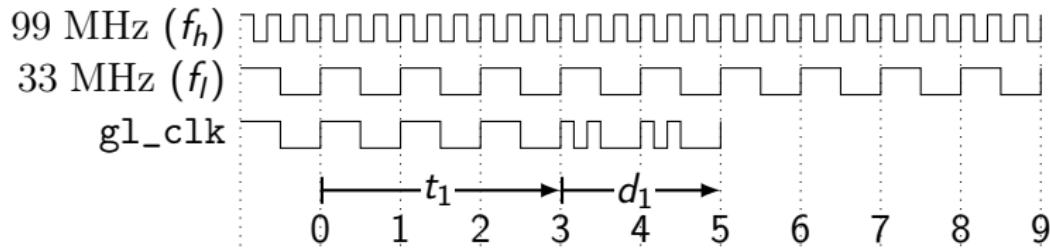
- Mechanism: CPU clock glitching
- Effect: Instruction skips



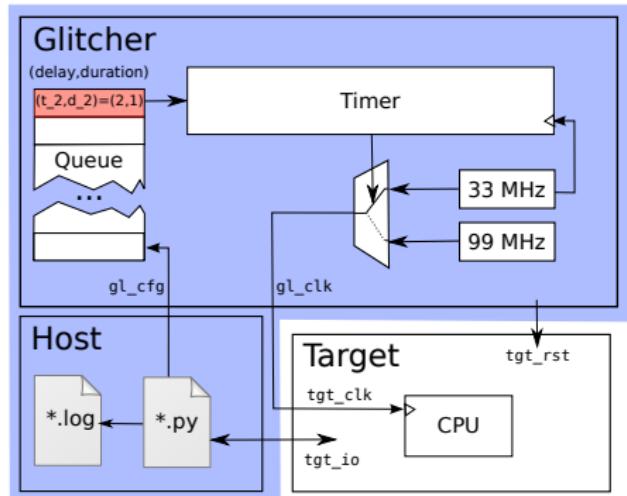
# Our attack: Schematic Hardware Setup



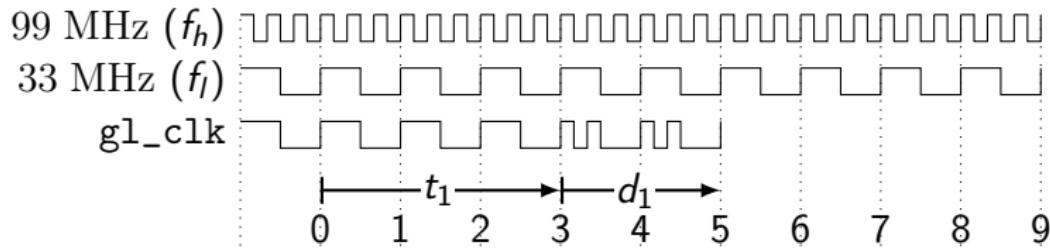
- Mechanism: CPU clock glitching
- Effect: Instruction skips



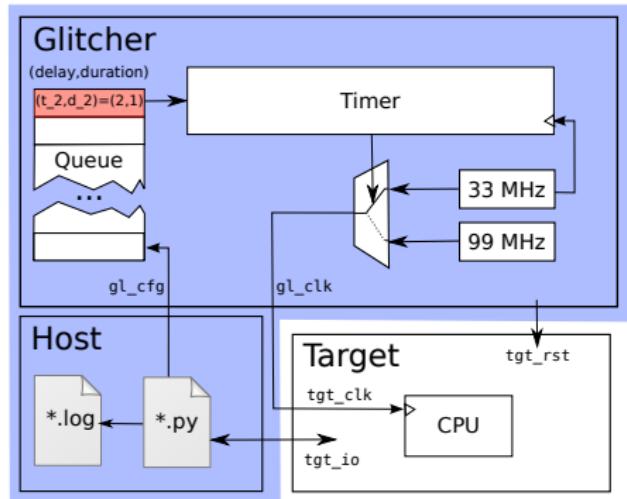
# Our attack: Schematic Hardware Setup



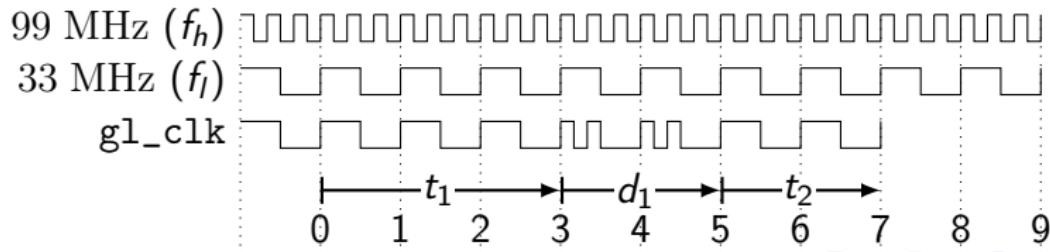
- Mechanism: CPU clock glitching
- Effect: Instruction skips



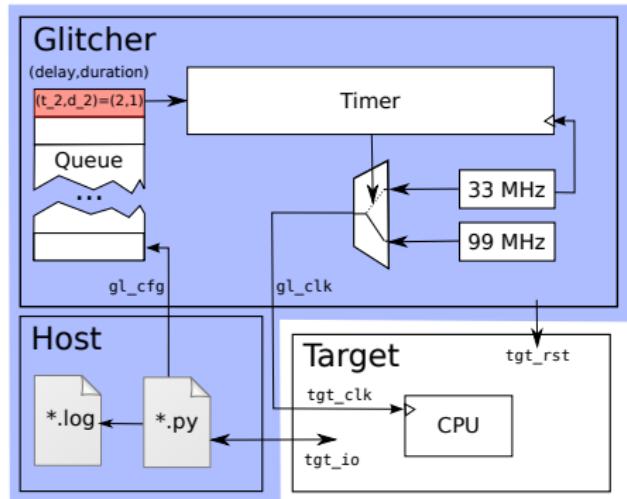
# Our attack: Schematic Hardware Setup



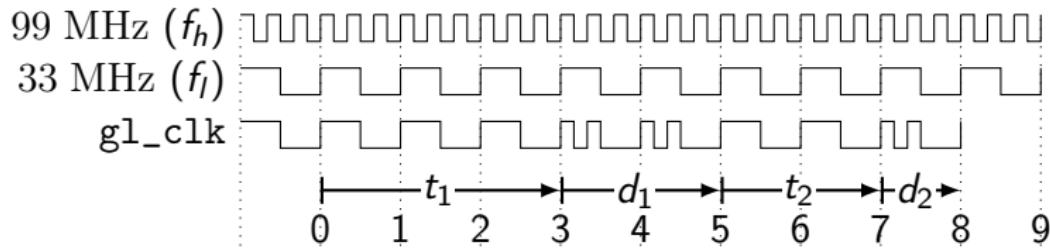
- Mechanism: CPU clock glitching
- Effect: Instruction skips



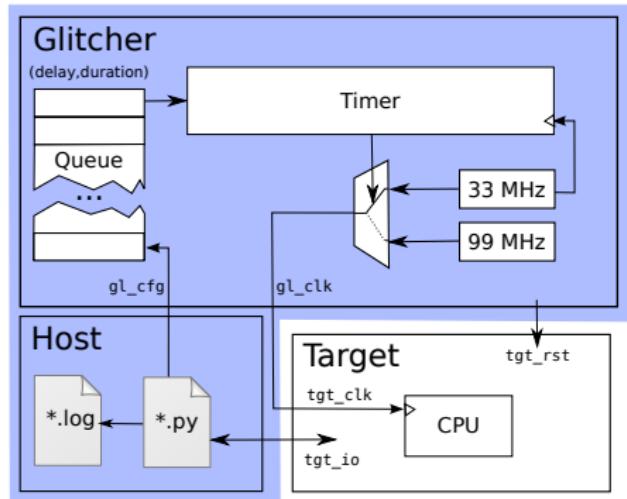
# Our attack: Schematic Hardware Setup



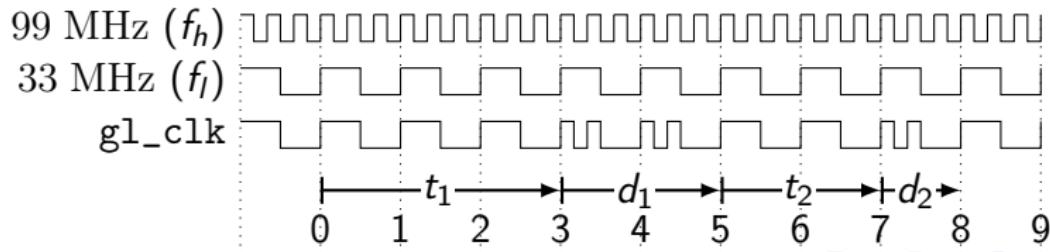
- Mechanism: CPU clock glitching
- Effect: Instruction skips



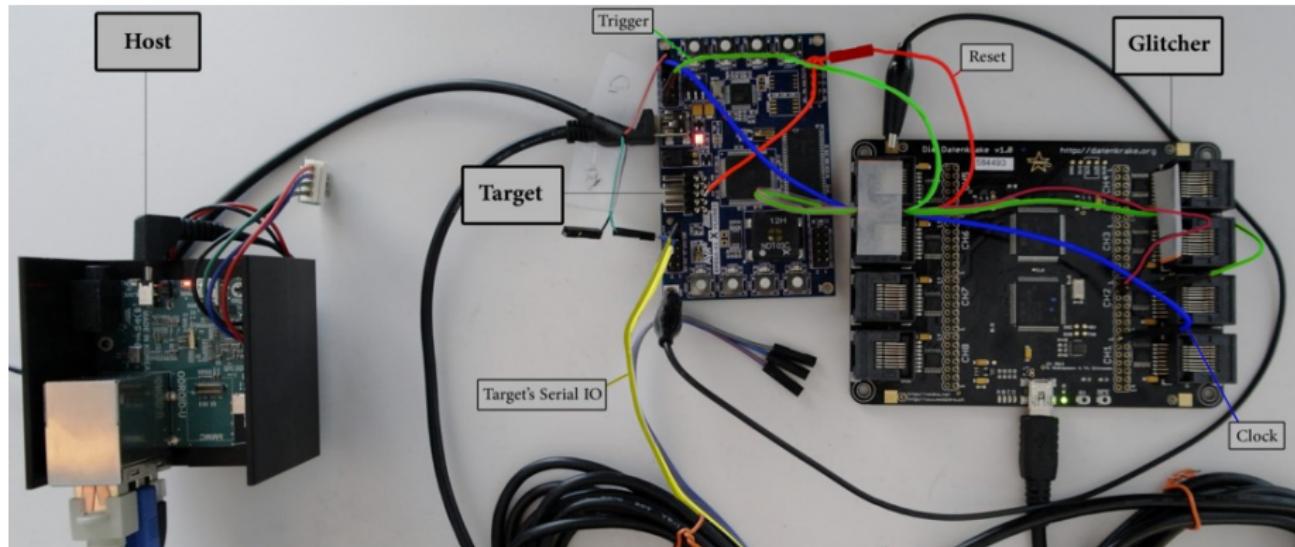
# Our attack: Schematic Hardware Setup



- Mechanism: CPU clock glitching
- Effect: Instruction skips



# Our attack: Real Hardware Setup



# Our target: Eta pairing of Relic toolkit on AVR



- Target hardware: Atmel AVR Xmega A1
- Target Software: Relic toolkit
  - Open source
  - Prime and Binary field arithmetic
  - Elliptic curves over prime and binary fields (NIST curves and pairing-friendly curves)
  - **Bilinear maps and related extension fields**
  - Cryptographic protocols
- Combination used on wireless sensor nodes as TinyPBC
- Unmodified code
  - No additional NOPs
  - No monitors
  - No triggers

# The RELIC implementation

**Input**  $P, Q \in \mathcal{E}$ ,  $r = (r_n \dots r_0)$

**Output**  $f_{r,P}(Q)$

```
1:  $T \leftarrow [2]P$ 
2:  $\alpha \leftarrow l_{P,P}(Q) \cdot l_{(r-1)P,P}(Q)$ 
3: for  $j \leftarrow n - 2, \dots, 1$  do
4:   if  $r_j = 1$  then
5:      $T \leftarrow T + P$ 
6:      $\alpha \leftarrow \alpha \cdot l_{T,P}(Q)$ 
7:   end if
8:    $T \leftarrow [2]T$ 
9:    $\alpha \leftarrow \alpha^2 \cdot l_{T,T}(Q)$ 
10: end for
11:  $\alpha \leftarrow \alpha^d$ 
12: return  $\alpha$ 
```

# The RELIC implementation

**Input**  $P, Q \in \mathcal{E}$ ,  $r = (r_n \dots r_0)$

**Output**  $f_{r,P}(Q)$

```

1:  $T \leftarrow [2]P$ 
2:  $\alpha \leftarrow l_{P,P}(Q) \cdot l_{(r-1)P,P}(Q)$ 
3: for  $j \leftarrow n - 2, \dots, 1$  do
4:   if  $r_j = 1$  then
5:      $T \leftarrow T + P$ 
6:      $\alpha \leftarrow \alpha \cdot l_{T,P}(Q)$ 
7:   end if
8:    $T \leftarrow [2]T$ 
9:    $\alpha \leftarrow \alpha^2 \cdot l_{T,T}(Q)$ 
10: end for
11:  $\alpha \leftarrow \alpha^d$ 
12: return  $\alpha$ 

```

```

.L2
...
call fb4_mul_dx
subi r16,1
sbc r17,__zero_reg__
breq .+2
rjmp .L2
subi r28,36
...
movw r22,r28
movw r24,r28
call etat_exp
pop r29
...

```

# The RELIC implementation

**Input**  $P, Q \in \mathcal{E}$ ,  $r = (r_n \dots r_0)$

**Output**  $f_{r,P}(Q)$

```

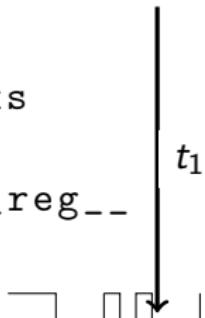
1:  $T \leftarrow [2]P$ 
2:  $\alpha \leftarrow l_{P,P}(Q) \cdot l_{(r-1)P,P}(Q)$ 
3: for  $j \leftarrow n - 2, \dots, 1$  do
4:   if  $r_j = 1$  then
5:      $T \leftarrow T + P$ 
6:      $\alpha \leftarrow \alpha \cdot l_{T,P}(Q)$ 
7:   end if
8:    $T \leftarrow [2]T$ 
9:    $\alpha \leftarrow \alpha^2 \cdot l_{T,T}(Q)$ 
10: end for
11:  $\alpha \leftarrow \alpha^d$ 
12: return  $\alpha$ 

```

```

.L2
...
call fb4_mul_dx
subi r16,1
sbc r17,__zero_reg__
breq .+2
rjmp .L2
subi r28,36
...
movw r22,r28
movw r24,r28
call etat_exp
pop r29
...

```



A timing diagram is shown on the right side of the assembly code. It consists of two parallel horizontal lines. A single rectangular pulse is drawn between them, starting at a point labeled  $t_1$  on the top line and ending at a point on the bottom line. This pulse corresponds to the execution of the instruction `breq .+2`.

# The RELIC implementation

**Input**  $P, Q \in \mathcal{E}$ ,  $r = (r_n \dots r_0)$

**Output**  $f_{r,P}(Q)$

```

1:  $T \leftarrow [2]P$ 
2:  $\alpha \leftarrow l_{P,P}(Q) \cdot l_{(r-1)P,P}(Q)$ 

4: if  $r_j = 1$  then
5:      $T \leftarrow T + P$ 
6:      $\alpha \leftarrow \alpha \cdot l_{T,P}(Q)$ 
7: end if
8:  $T \leftarrow [2]T$ 
9:  $\alpha \leftarrow \alpha^2 \cdot l_{T,T}(Q)$ 

11:  $\alpha \leftarrow \alpha^d$ 
12: return  $\alpha$ 

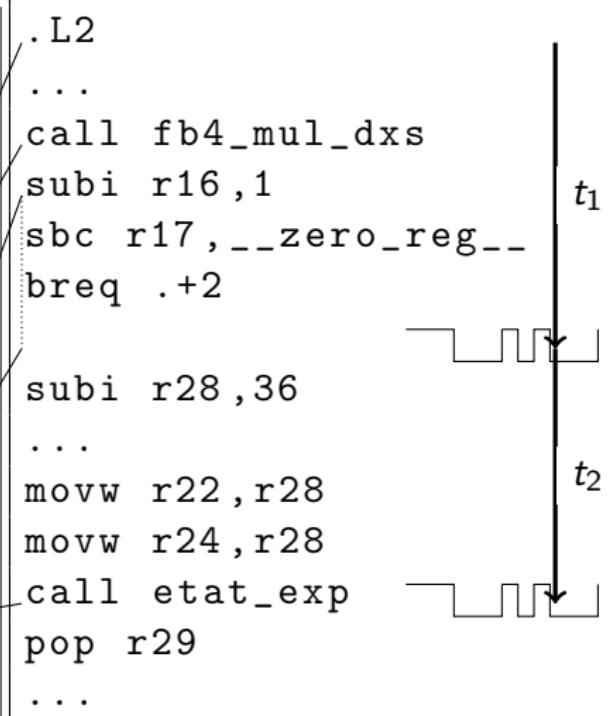
```

```

.L2
...
call fb4_mul_dx
subi r16,1
sbc r17,__zero_reg__
breq .+2

subi r28,36
...
movw r22,r28
movw r24,r28
call etat_exp
pop r29
...

```



# The RELIC implementation

**Input**  $P, Q \in \mathcal{E}$ ,  $r = (r_n \dots r_0)$

**Output**  $f_{r,P}(Q)$

```

1:  $T \leftarrow [2]P$ 
2:  $\alpha \leftarrow l_{P,P}(Q) \cdot l_{(r-1)P,P}(Q)$ 

4: if  $r_j = 1$  then
5:    $T \leftarrow T + P$ 
6:    $\alpha \leftarrow \alpha \cdot l_{T,P}(Q)$ 
7: end if
8:  $T \leftarrow [2]T$ 
9:  $\alpha \leftarrow \alpha^2 \cdot l_{T,T}(Q)$ 

11:
12: return  $\alpha$ 

```

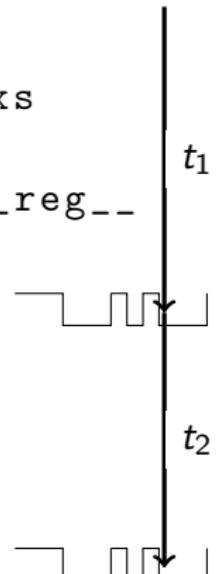
```

.L2
...
call fb4_mul_dx
subi r16,1
sbc r17,__zero_reg__
breq .+2

subi r28,36
...
movw r22,r28
movw r24,r28

pop r29
...

```



# The RELIC implementation

**Input**  $P, Q \in \mathcal{E}$ ,  $r = (r_n \dots r_0)$

**Output**  $f_{r,P}(Q)$

```

1:  $T \leftarrow [2]P$ 
2:  $\alpha \leftarrow l_{P,P}(Q) \cdot l_{(r-1)P,P}(Q)$ 

4: if  $r_j = 1$  then
5:    $T \leftarrow T + P$ 
6:    $\alpha \leftarrow \alpha \cdot l_{T,P}(Q)$ 
7: end if
8:  $T \leftarrow [2]T$ 
9:  $\alpha \leftarrow \alpha^2 \cdot l_{T,T}(Q)$ 

11:
12: return  $\alpha$ 

```

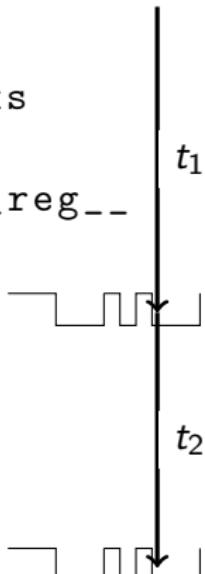
```

.L2
...
call fb4_mul_dx
subi r16,1
sbc r17,__zero_reg__
breq .+2

subi r28,36
...
movw r22,r28
movw r24,r28

pop r29
...

```



# Host: Outline of Analysis

- ① Output with successful glitch:

$$\beta' = (l_{P,P}(Q) \cdot l_{(n-1)P,P}(Q))^2 \cdot l_{2P,2P}(Q)$$

- ② Capture secret as root of polynomial of degree 5:

$$f(x, y) = \beta' - (l_{P,P}(x, y) \cdot l_{(r-1)P,P}(x, y))^2 \cdot l_{2P,2P}(x, y)$$

- ③ Compute simultaneous roots of  $f(x, y)$  and  $\mathcal{E} : y^2 = x^3 - x$
- ④ Test candidates  $Q'$  against result of correct pairing:

$$e(P, Q') = e(P, Q)?$$

# Timing of first fault is critical

## The challenge for 2nd order attack

- The timings  $t_1, t_2$  of the target instructions depend on unknown secret (e.g.  $Q$ )
- It is not possible to detect case where only one glitch is successful
- Many combinations have to be tested

## Our strategy

- ➊ Profiling: Determine probability distribution of  $t_1$  and  $t_2$  for randomized secret input (e.g.  $Q$ )
- ➋ Attack:
  - Rank candidates for  $t_1$  and  $t_2$  according to their probability
  - Introduce fault as early as possible
- ➌ Analysis: Full Automation

# Host: Outline of Analysis

- ① Output with successful glitch:

$$\beta' = (l_{P,P}(Q) + l_{2P,2P}(Q)) \cdot (l_{P,P}(Q) - l_{2P,2P}(Q))^2 \cdot l_{2P,2P}(Q)$$

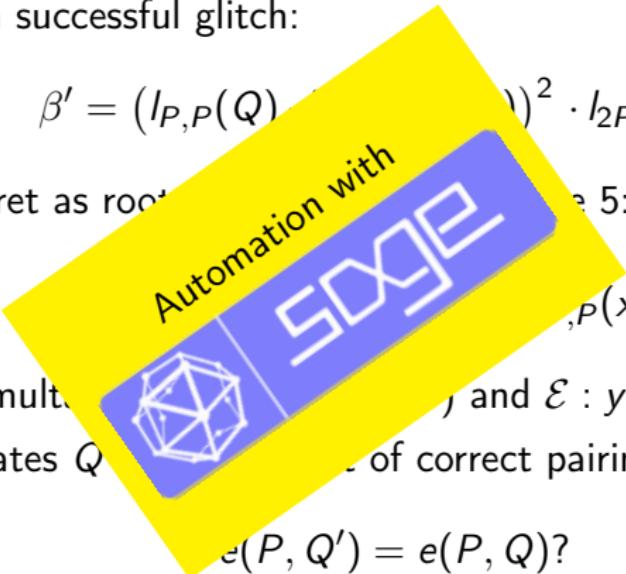
- ② Capture secret as root of unity and compute discrete log: 5:

$$f(x, y) = (l_{P,P}(x, y) + l_{2P,2P}(x, y))^2 \cdot l_{2P,2P}(x, y)$$

- ③ Compute simultaneous discrete logarithms for  $P$  and  $E : y^2 = x^3 - x$

- ④ Test candidates  $Q'$  for being a correct pairing:

$$e(P, Q') = e(P, Q)?$$

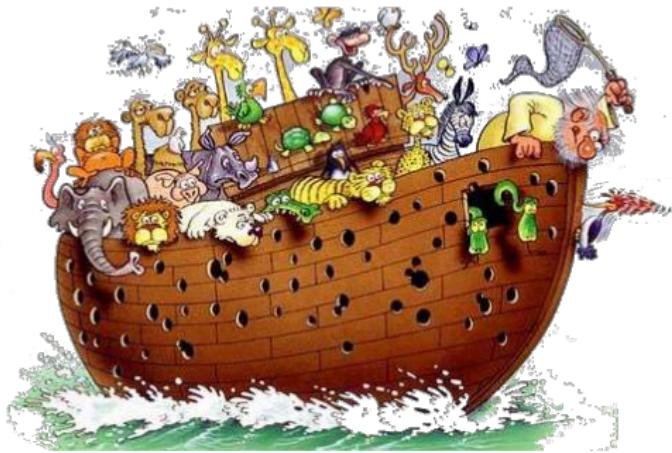


# Performance of the attack

- In < 10 seconds per experiment (average):
  - Self-tests
  - Configure glitcher
  - Restart target
  - Induce faults
  - Analyze result
- More than 10000 experiments per day

# Conclusion

- Second order attacks on pairings possible
- Two stage computation: not enough protection
- Add dedicated countermeasures as protection against **active** attacks



# Simplifiy inversion of final exponentiation with faults

Ongoing work

## Problem

Final exponentiation cannot always be skipped:

- Inlining at higher optimization levels
- Countermeasures that guarantee execution of function

## Example (Inlining at higher optimization levels)

```
...  
movw r22,r28  
movw r24,r28  
call etat_exp  
pop r29  
...
```

```
...  
movw r22,r28  
movw r24,r28  
jmp etat_exp  
nop  
...
```

# Simplifiy inversion of final exponentiation with faults

Ongoing work

## Our approach

- Skip part of final exponentiation to modify exponent:

$$d = (p^k - 1)/r \rightarrow d' = (p^k - 1)/r + \delta$$

- Simplify mathematical inversion of final exponent  $d'$

# References

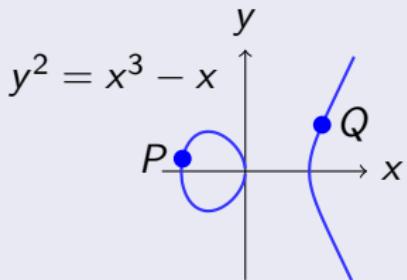
- Relic toolkit:  
<http://code.google.com/p/relic-toolkit/>
- Glitcher Die Datenkrake:  
[https://www.usenix.org/conference/woot13/  
workshop-program/presentation/nedospasov](https://www.usenix.org/conference/woot13/workshop-program/presentation/nedospasov)

# Background

## The basic building block

Bilinear mapping:

$$\begin{aligned} e : \mathcal{E}(\mathbb{F}_{p^k}) \times \mathcal{E}(\mathbb{F}_{p^k}) &\rightarrow \mathbb{F}_{p^k}^* \\ (P, Q) &\mapsto f_{n,P}(Q)^d \end{aligned}$$



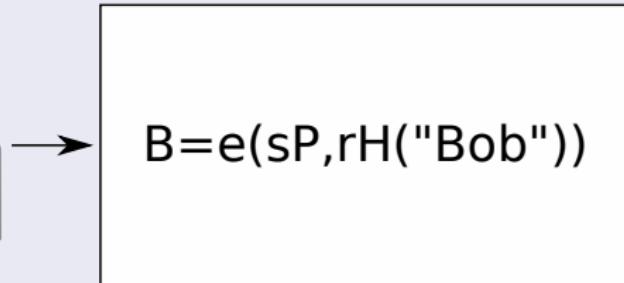
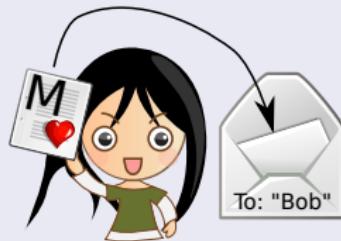
- $n, d$  are huge
- $f_{n,P}(x, y)$ : zero of order  $n$  at  $P$ , degree  $> n$

## Interesting properties for application in cryptography

- Bilinearity:  $e(aP, bQ) = e(P, Q)^{ab} = e(bP, aQ)$
- Hard to invert
- $f_{n,P}(Q)$  is efficiently computable with Miller algorithm

# An example Application: IBE

## Encryption



## Decryption



# An example Application: IBE

## Encryption



$$B = e(sP, rH("Bob"))$$



The secret decryption key  
is one argument of the  
pairing.

## Decryption



<rP, M\*B>



$$M*B/e(rP, sH("bob"))$$



# Miller Algorithm (Victor Miller 1986)

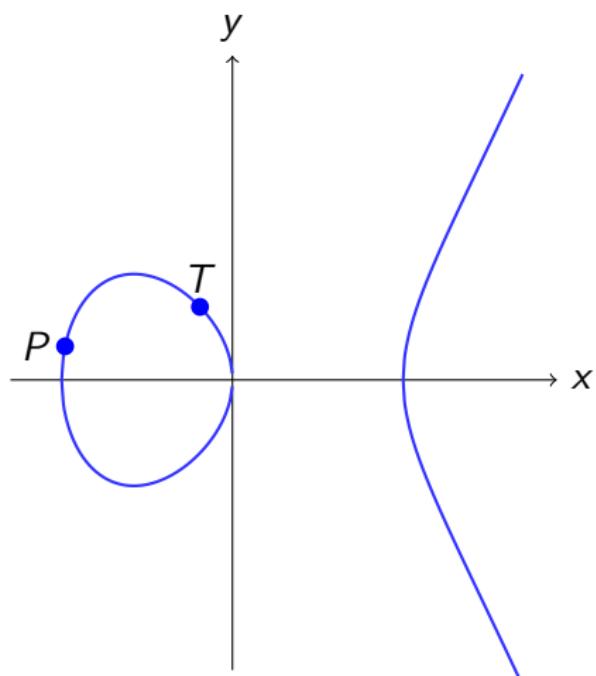
Extending the elliptic curve double and add algorithm

$$y^2 = x^3 - x$$

**Input**  $P, Q \in \mathcal{E}$ ,  $n = (n_{t-1} \dots n_0)$

**Output**  $f_{n,P}(Q)$

```
1:  $\alpha \leftarrow 1, T \leftarrow P$ 
2: for  $j \leftarrow t-2, \dots, 0$  do
3:    $\alpha \leftarrow \alpha^2 \cdot l_{T,T}(Q) / v_{2T}(Q)$ 
4:    $T \leftarrow 2T$ 
5:   if  $n_j = 1$  then
6:      $\alpha \leftarrow \alpha \cdot l_{T,P}(Q) / v_{T+P}(Q)$ 
7:      $T \leftarrow T + P$ 
8:   end if
9: end for
10: return  $\alpha^d$ 
```



# Miller Algorithm (Victor Miller 1986)

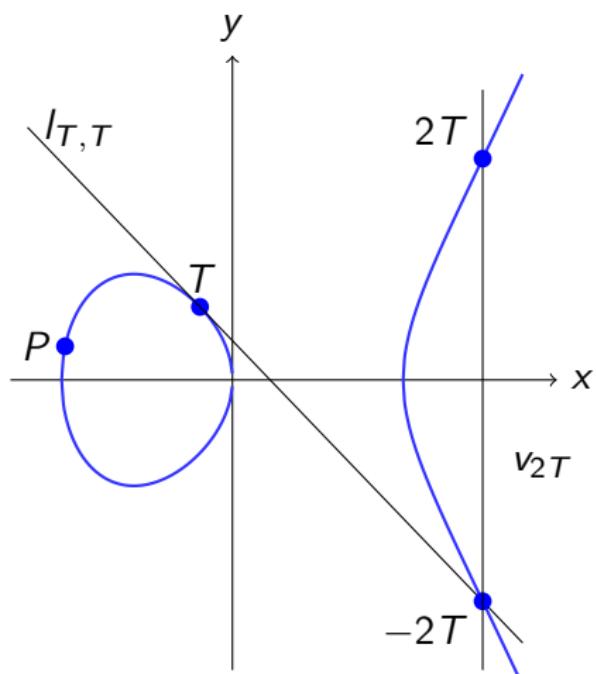
Extending the elliptic curve double and add algorithm

$$y^2 = x^3 - x$$

**Input**  $P, Q \in \mathcal{E}$ ,  $n = (n_{t-1} \dots n_0)$

**Output**  $f_{n,P}(Q)$

```
1:  $\alpha \leftarrow 1, T \leftarrow P$ 
2: for  $j \leftarrow t-2, \dots, 0$  do
3:    $\alpha \leftarrow \alpha^2 \cdot I_{T,T}(Q) / v_{2T}(Q)$ 
4:    $T \leftarrow 2T$ 
5:   if  $n_j = 1$  then
6:      $\alpha \leftarrow \alpha \cdot I_{T,P}(Q) / v_{T+P}(Q)$ 
7:      $T \leftarrow T + P$ 
8:   end if
9: end for
10: return  $\alpha^d$ 
```



# Miller Algorithm (Victor Miller 1986)

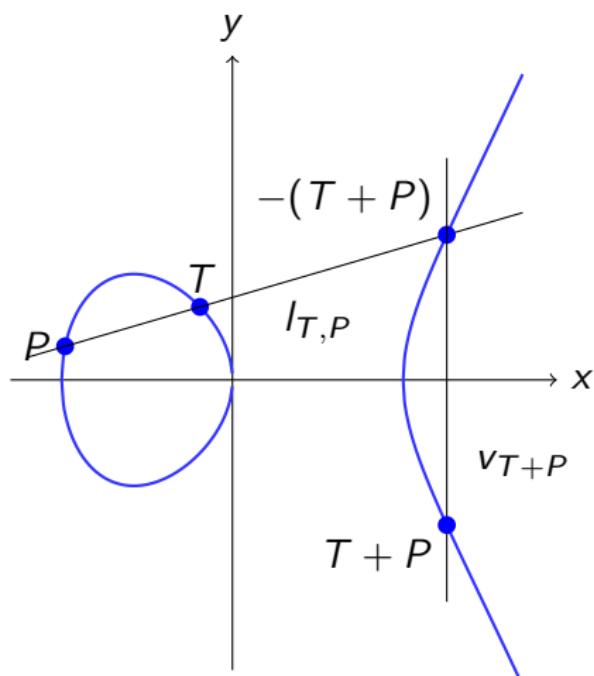
Extending the elliptic curve double and add algorithm

$$y^2 = x^3 - x$$

**Input**  $P, Q \in \mathcal{E}$ ,  $n = (n_{t-1} \dots n_0)$

**Output**  $f_{n,P}(Q)$

```
1:  $\alpha \leftarrow 1, T \leftarrow P$ 
2: for  $j \leftarrow t-2, \dots, 0$  do
3:    $\alpha \leftarrow \alpha^2 \cdot I_{T,T}(Q) / v_{2T}(Q)$ 
4:    $T \leftarrow 2T$ 
5:   if  $n_j = 1$  then
6:
7:      $\alpha \leftarrow \alpha \cdot I_{T,P}(Q) / v_{T+P}(Q)$ 
8:      $T \leftarrow T + P$ 
9:   end if
10:  end for
11: return  $\alpha^d$ 
```



## Different delays/instructions, same effect

```
.L2
...
call fb4_mul_dx           //sets zero flag: Z=1
subi r16,1                //re-sets zero flag: Z=0
sbc r17,__zero_reg__     //Z=Z
breq .+2                  //Z=1: branch
rjmp .L2
subi r28,36
...
```

## Different delays/instructions, same effect

```
.L2
...
call fb4_mul_dx           //sets zero flag: Z=1
subi r16,1                //re-sets zero flag: Z=0
sbc r17,__zero_reg__     //Z=Z
breq .+2                  //Z=1: branch

subi r28,36
...
```

## Different delays/instructions, same effect

```
.L2
...
call  fb4_mul_dx           //sets zero flag: Z=1
subi  r16,1                //re-sets zero flag: Z=0
sbc   r17,__zero_reg__    //Z=Z
breq .+2                  //Z=1: branch
rjmp  .L2
subi  r28,36
...
```

## Different delays/instructions, same effect

```
.L2
...
call fb4_mul_dx           //sets zero flag: Z=1

sbc r17,__zero_reg__    //Z=Z
breq .+2                 //Z=1: branch
rjmp .L2
subi r28,36
...
```

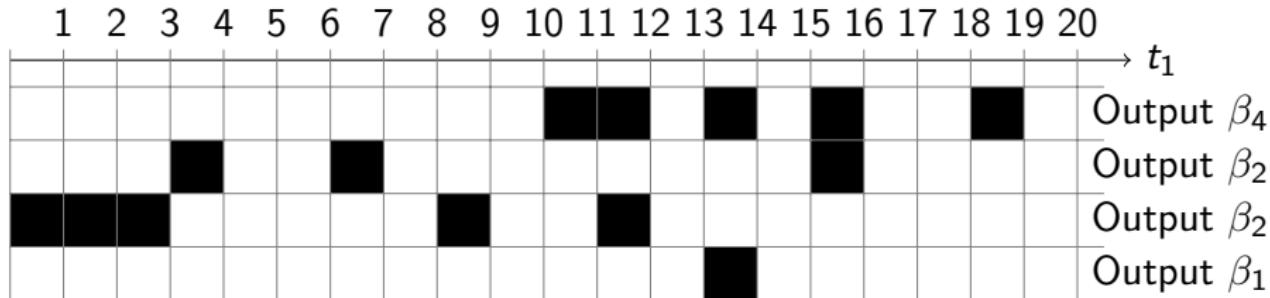
# 1st order attack

Group delays by output and locate rjmp .L2

subi rjmp .L2



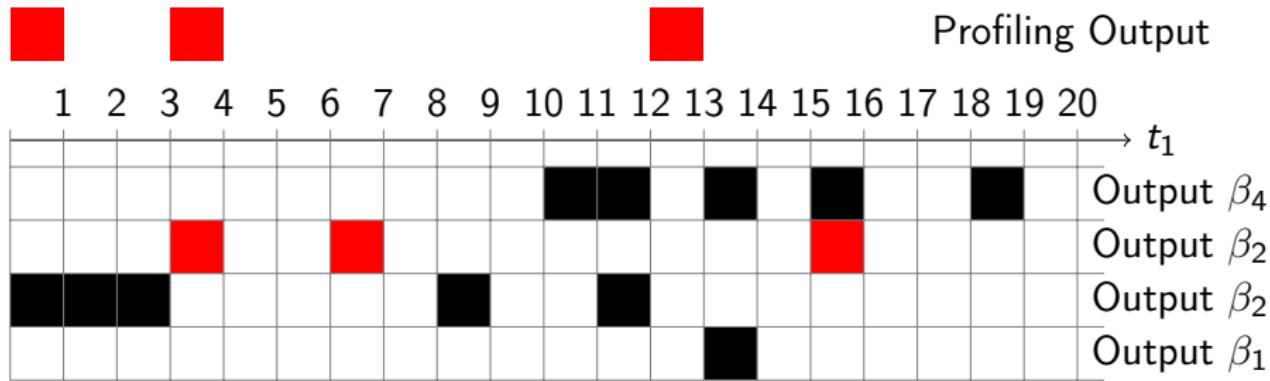
Profiling Output



# 1st order attack

Group delays by output and locate `rjmp .L2`

`subi rjmp .L2`



- Output  $\beta_2$  matches pattern of profiling
- $\Rightarrow t_1 = 6$  is instruction of `rjmp .L2`
- Proceed with 2nd order attack and correct setting of  $t_1$