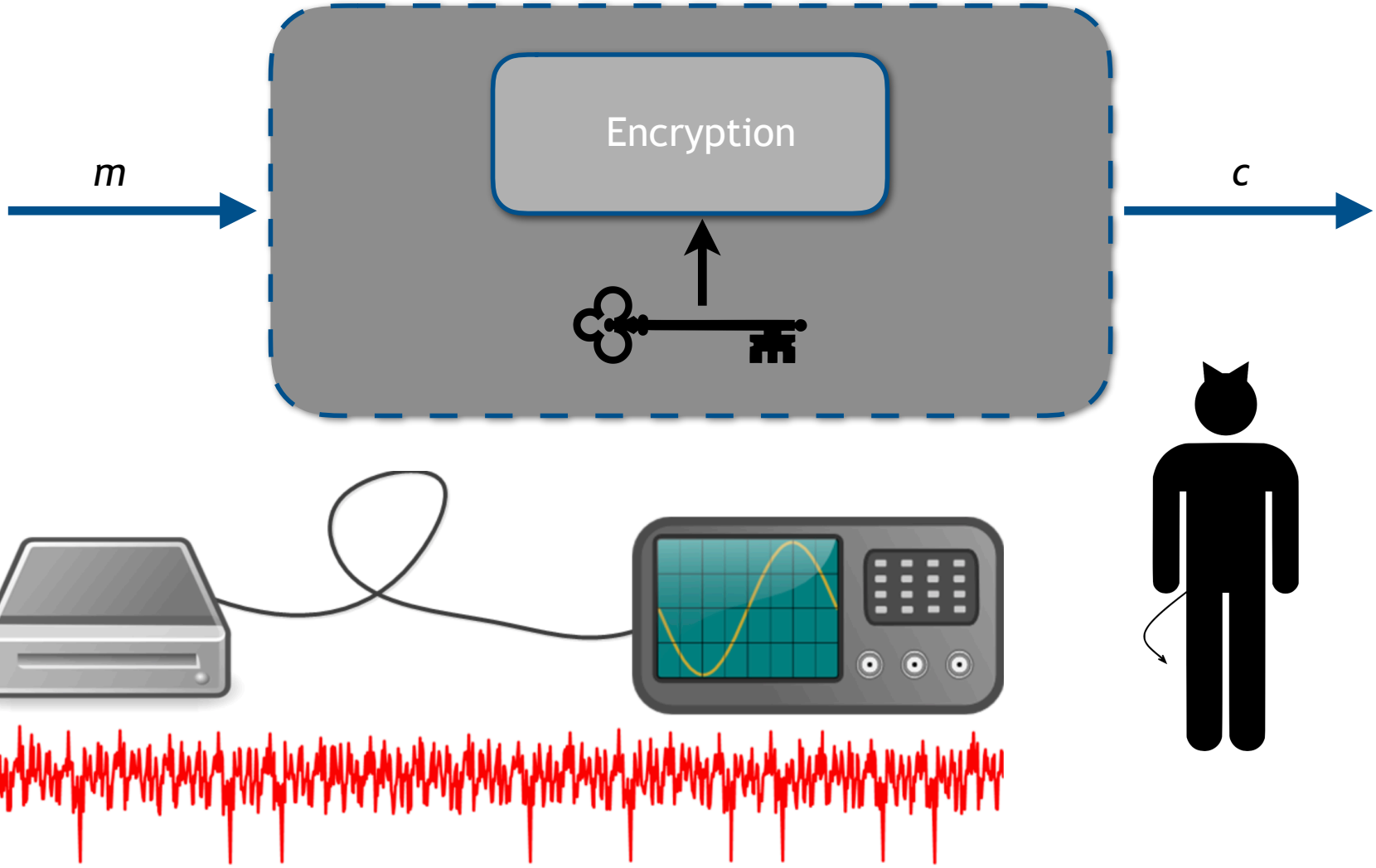
Protecting the most significant bits on scalar multiplication algorithms

Estuardo Alpirez Bock, Lukasz Chmielewski and
Konstantina Miteloudi from Radboud University

SCA



Outline

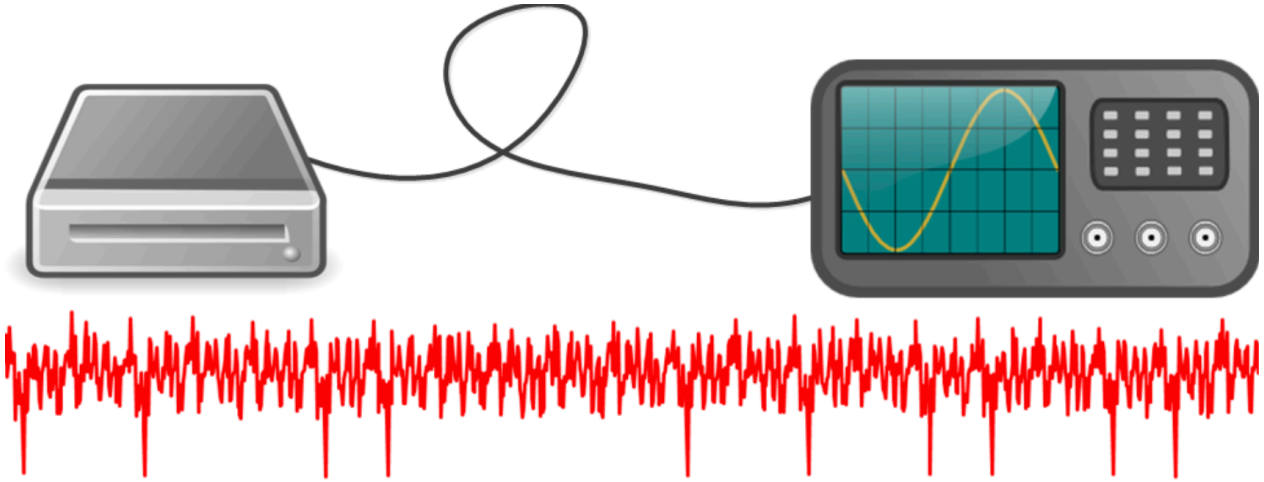
- (Simple) Side channel analysis
- Scalar multiplication on ECC
- Leakage in the beginning of the multiplication
  - Previous results
  - Our results (leakage on Curve25519 and Complete addition formulas)
- Removing the leakage
- Conclusions

Side Channel Analysis Attacks

SCA attacks are *passive* physical attacks based on the observation of a hardware device during the execution of cryptographic operations

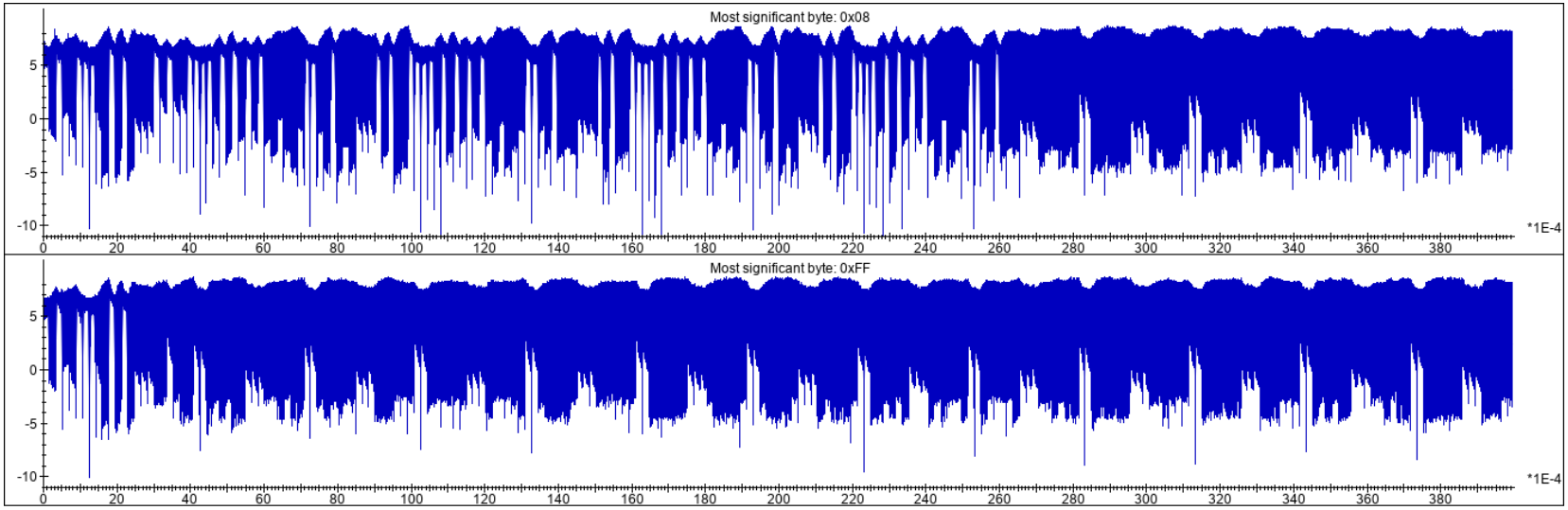
Parameters observed:

- Power consumption (power analysis)
- Electromagnetic radiation (EM analysis)
- Execution times (timing attacks)



Simple SCA

- Usually one measurement is enough for extracting secrets from implementations



Scalar multiplication

- Used in ECC
- But also in isogeny based cryptographic schemes, e.g. SIKE, for calculating a secret kernel

- The multiplication is performed using a secret scalar *k*, for adding a point *P* on the curve

$$kP = \underbrace{P + P + \ldots + P}_{k \text{ times}}$$

- Security on ECC Given *P* and *kP*, it should be difficult to obtain *k*

Montgomery Ladder

Input: $k = (k_{l-1}, ..., k_1, k_0)_2$ with $k_{l-1} = 1, P = (x, y) \in E(GF(2^m))$.

Output: kP.

$Q[0] \leftarrow P$, $Q[1] \leftarrow 2P$.

for i from $l - 2$ downto 0 **do**

  if $k_i = 1$ **then**

    $Q[0] \leftarrow Q[0] + Q[1]$, $Q[1] \leftarrow 2Q[1]$.

  else

    $Q[1] \leftarrow Q[0] + Q[1]$, $Q[0] \leftarrow 2Q[0]$.

  end if

end for

return $Q[0]$.

Montgomery Ladder: robust against SPA

Input: $k = (k_{l-1}, ..., k_1, k_0)_2$ with $k_{l-1} = 1, P = (x, y) \in E(GF(2^m))$.

Output: kP.

$Q[0] \leftarrow P$, $Q[1] \leftarrow 2P$.

for i from $l-2$ downto 0 **do**

  if $k_i = 1$ **then**

    $Q[0] \leftarrow Q[0] + Q[1]$, $Q[1] \leftarrow 2Q[1]$.

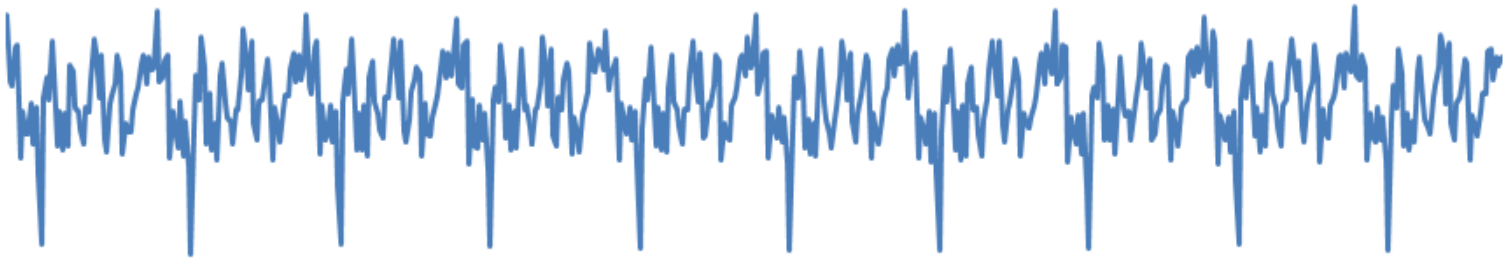
  else

    $Q[1] \leftarrow Q[0] + Q[1]$, $Q[0] \leftarrow 2Q[0]$.

  end if

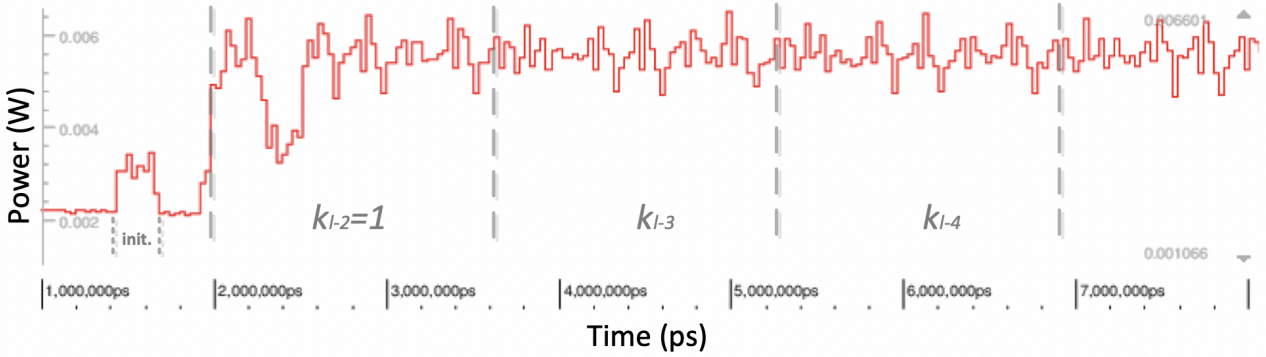
end for

return $Q[0]$.

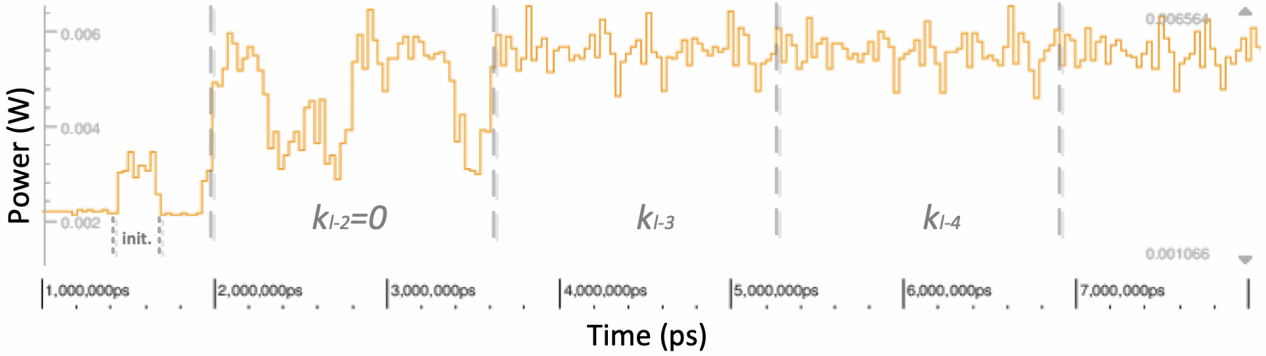


Leakage in implementations using Lopez-Dahab projective coordinates

Previous works [1,2] showed that the second bit of the scalar could be easily extracted via SPA on implementations using LD-projective coordinates



(a)



[1]: Alpirez Bock et al.: Increasing the robustness of the Montgomery kP-Aglorithm by modifying its initialisation, SECITC 2016

[2]: Aranha et al.: LadderLeak, CCS 2020

Montgomery Ladder using Lopez-Dahab coordinates

Input: $k = (k_{l-1}, ..., k_1, k_0)_2$ with $k_{l-1} = 1, P = (x, y) \in E(GF(2^m))$.
Output: $kP = (x_1, y_1)$.

1: $X_1 \leftarrow x$, $Z_1 \leftarrow 1$, $X_2 \leftarrow x^4 + b$, $Z_2 \leftarrow x^2$.
2: **for** i from $l-2$ downto 0 **do**
3: **if** $k_i = 1$ **then**
4: $T \leftarrow Z_1$, $Z_1 \leftarrow (X_1 Z_2 + X_2 Z_1)^2$, $X_1 \leftarrow xZ_1 + X_1 X_2 T Z_2$,
5: $T \leftarrow X_2$, $X_2 \leftarrow X_2^4 + bZ_2^4$, $Z_2 \leftarrow T^2 Z_2^2$.
6: **else**
7: $T \leftarrow Z_2$, $Z_2 \leftarrow (X_2 Z_1 + X_1 Z_2)^2$, $X_2 \leftarrow xZ_2 + X_1 X_2 T Z_1$,
8: $T \leftarrow X_1$, $X_1 \leftarrow X_1^4 + bZ_1^4$, $Z_1 \leftarrow T^2 Z_1^2$.
9: **end if**
10: **end for**
11: $x_1 \leftarrow X_1/Z_1$.
12: $y_1 \leftarrow y + (x + x_1)[X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1 Z_2)]/(xZ_1 Z_2)$.
13: **return** $((x_1, y_1))$.

LadderLeak Attack

- In [2] the authors identified this leakage on recent versions of OpenSSL, which implements ECDSA using the Lopez-Dahab projective coordinates
  - The leakage is identified in the form of cache attacks

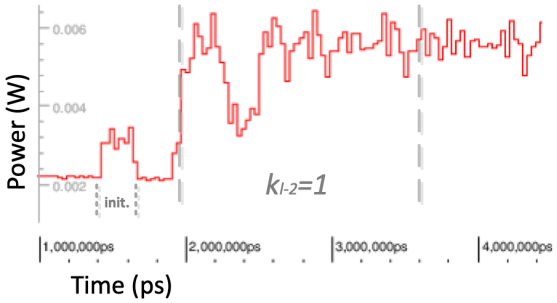
- The authors even show how this leakage could be exploited for solving the *Hidden numbers problem*, leading to a full key recovery
  - HNP shows that the MSBs of a secret key are as hard to guess as the entire key [3]
  - [3] presents an algorithm for recovering the key, given some bits of leakage

[2]: Aranha et al.: LadderLeak, CCS 2020

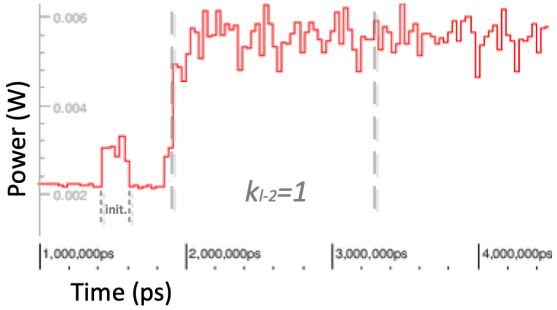
[3]: Boneh et al.: Hardness of computing the MSBs of secret keys in Diffie-Hellman and related schemes, CRYPTO 96

Mitigation of the leakage

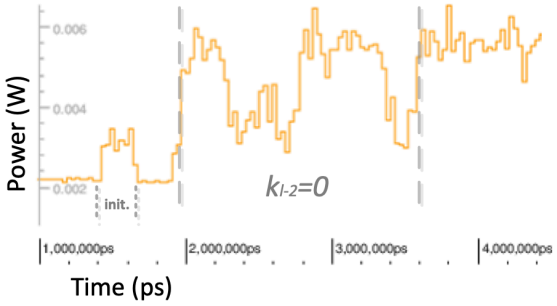
- [1] proposes a re-design of the initialisation phase of the algorithm and show how their re-design effectively mitigates this leakage, without implying any additional efficiency or size costs



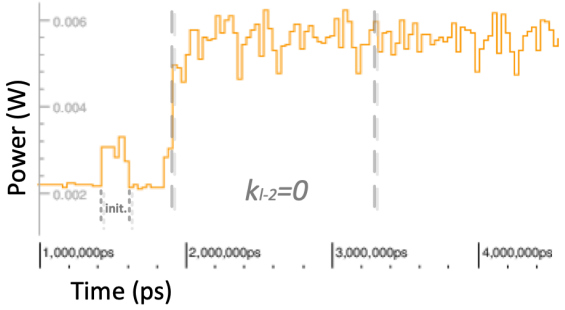
(a) $k1 \cdot P$ according to Algorithm 1



(c) $k1 \cdot P$ according to Algorithm 2



(b) $k2 \cdot P$ according to Algorithm 1



(d) $k2 \cdot P$ according to Algorithm 2

[1]: Alpirez Bock et al.: Increasing the robustness of the Montgomery kP-Aglorithm by modifying its initialisation, SECITC 2016

Leakage at the beginning

We observe that many implementations of the Montgomery ladder easily leak the MSBs of the secret scalar

- Consider 2 case studies:
  - Software implementation of Curve25519 from [Ba+21]
  - Hardware implementation of the Complete Addition Formulas from [Pi+18]
- Show the presence of this leakage on each implementation, analyse it and propose a re-design of part of the algorithm
- (Some) of our re-designs barely imply efficiency penalties

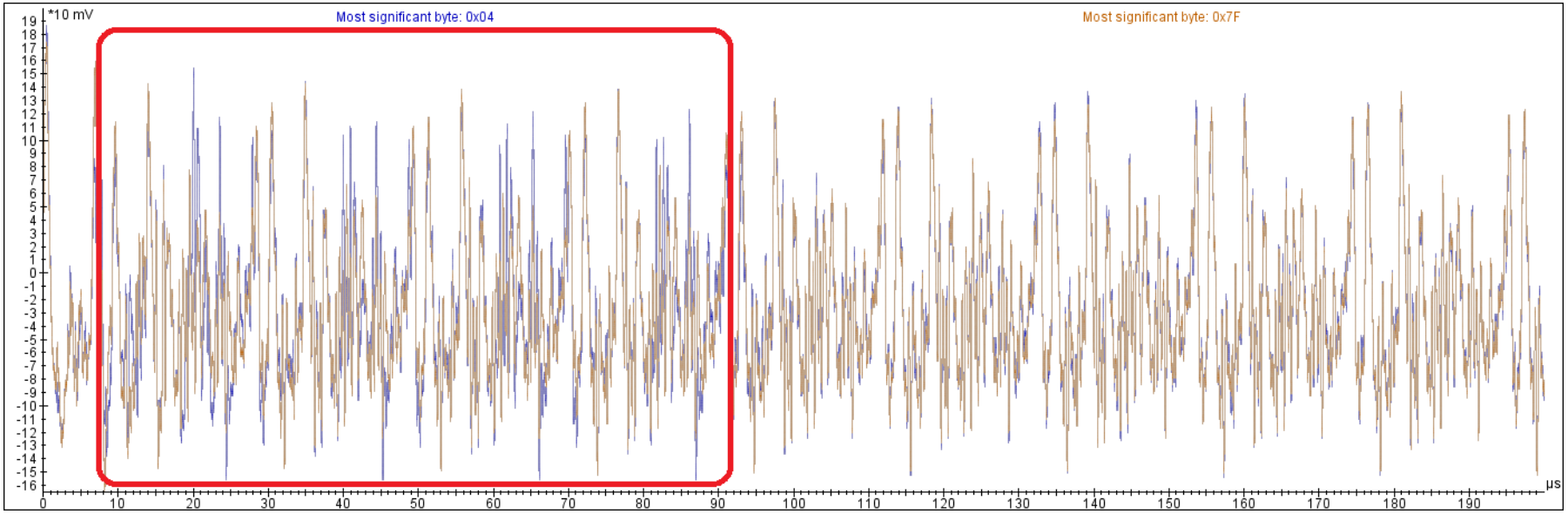
[Ba+21]: Batina et al.: SCA-secure ECC in software – mission impossible?, to appear in CHES23

[Pi+18]: Pirotte et al.: Design of a Fully Balanced ASIC Coprocessor Implementing Complete Addition Formulas on Weierstrass Elliptic Curves, DSD 2018

Leakage on Cruve25519

Compare the implementation ran on two different keys:

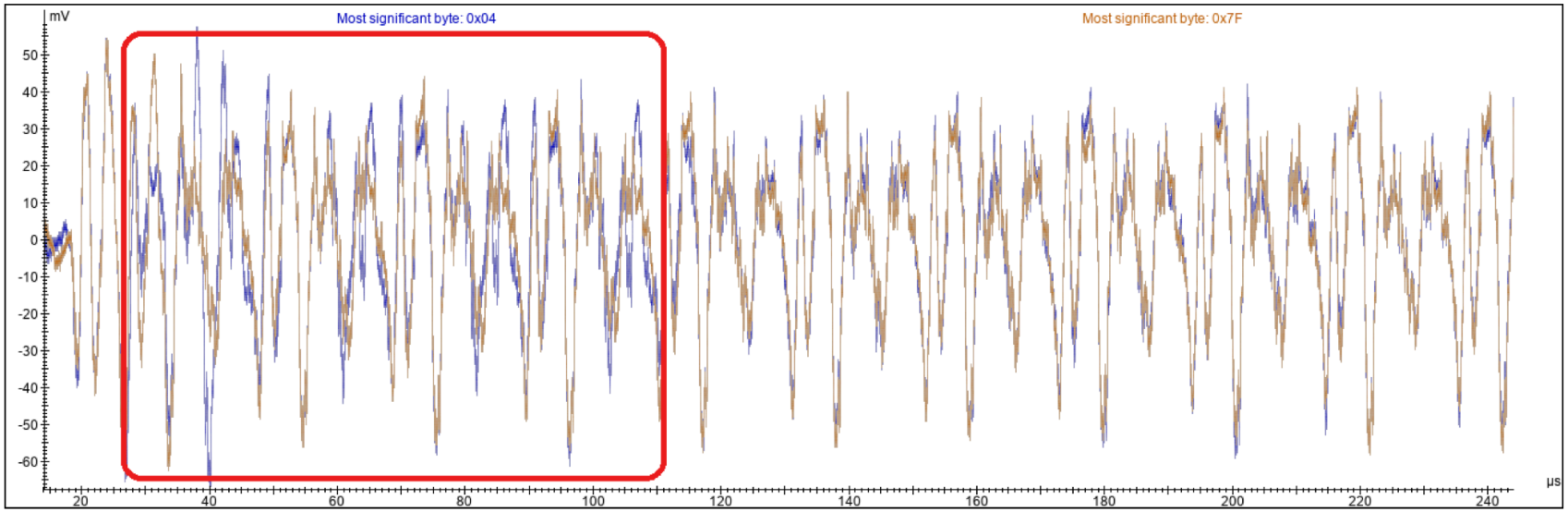
$$k_1 = 00000100$$
$$k_2 = 01111111$$



Leakage on Cruve25519 using Z-coordinate randomisation

Compare the implementation ran on two different keys:

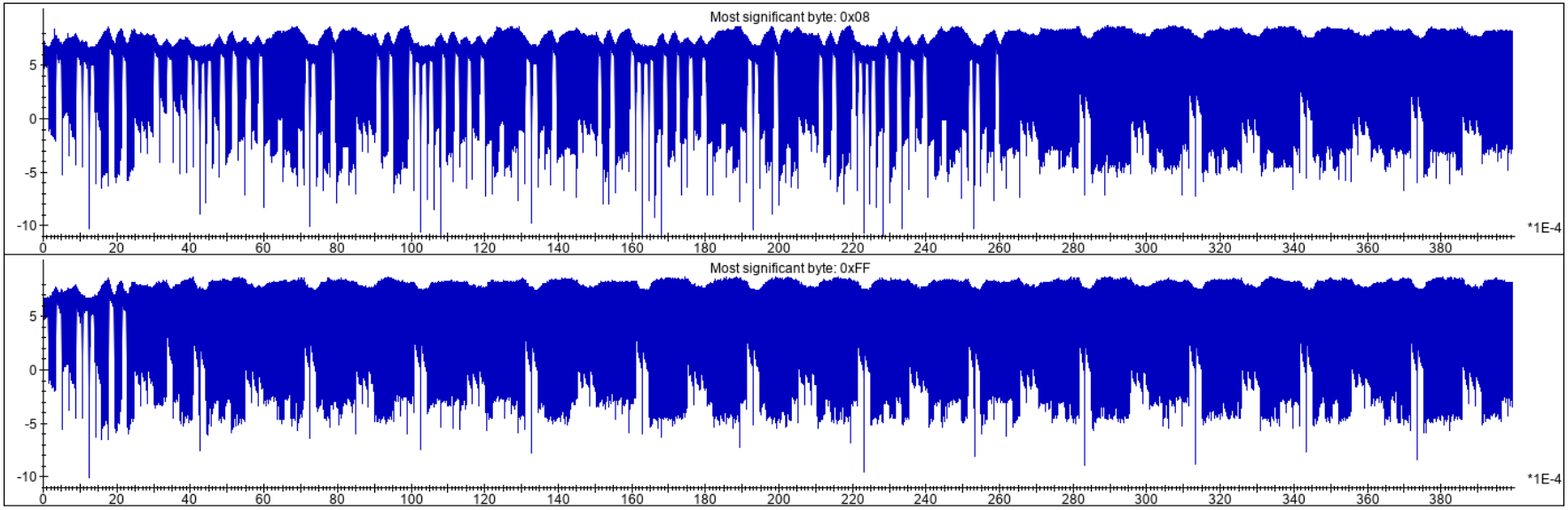
$$k_1 = 00000100$$
$$k_2 = 01111111$$



Leakage on the complete addition formulas

Compare the implementation ran on two different keys:

$$k_1 = 00000100$$
$$k_2 = 01111111$$



Reason for the leakage

Algorithm 2 Montgomery Ladder for x-coordinate-based scalar multiplication on $E : y^2 = x^3 + 486662x^2 + x$ [4]

Inputs: $k \in \{0, \ldots, 2^{255} - 1\}, x_P$

$X_1 \leftarrow 1$; $Z_1 \leftarrow 0$; $X_2 \leftarrow x_P$; $Z_2 \leftarrow 1$; $p \leftarrow 0$

for $i \leftarrow 254$ **downto** 0 **do**

  $c \leftarrow k[i] \oplus p$; $p \leftarrow k[i]$ ▷ $k[i]$ denotes bit i of k

  $(X_1, Z_1, X_2, Z_2) \leftarrow \mathsf{cswap}(X_1, Z_1, X_2, Z_2, c)$

  $(X_1, Z_1, X_2, Z_2) \leftarrow \mathsf{ladderstep}(x_p, X_1, Z_1, X_2, Z_2)$

end for

return (X_1, Z_1)

Different number of multiplications times 0 or times 1 will be performed depending on the value of the MSB(s).

Reason for the leakage

Different number of multiplications times 0 or times 1 will be performed depending on the value of the MSB(s).

Algorithm 4 Ladder step when entered just after the initialisation for $k[0] = 0$

Inputs: $x_P, X_1 = 1, Z_1 = 0, X_2 = x_p, Z_2 = 1$

1: $T_1 \leftarrow x_P + 1$
2: $X_2 \leftarrow x_P - 1$
3: $Z_2 \leftarrow 1 + 0$
4: $X_1 \leftarrow 1 - 0$
5: $T_1 \leftarrow T_1 \cdot 1$
6: $X_2 \leftarrow X_2 \cdot 1$
7: $Z_2 \leftarrow 1 \cdot 1$
8: $X_1 \leftarrow 1 \cdot 1$
9: $T_2 \leftarrow 1 - 1 = 0$
10: $Z_1 \leftarrow 0 \cdot a24$
11: $Z_1 \leftarrow 0 + 1$
12: $Z_1 \leftarrow 0 \cdot 1$
13: $X_1 \leftarrow 1 \cdot 1$
14: $Z_2 \leftarrow (x_P + 1) - (x_P - 1) = 2$
15: $Z_2 \leftarrow 2 \cdot 2 = 4$
16: $Z_2 \leftarrow 4 \cdot x_P$
17: $X_2 \leftarrow (x_P + 1) + (x_P - 1) = 2x_P$
18: $X_2 \leftarrow 2x_P \cdot 2x_P = 4x_P^2$
return $(X_1 = 1, Z_1 = 0);$
$(X_2 = 4x_P^2, Z_2 = 4x_P)$

Reason for the leakage

Different number of multiplications times 0 or times 1 will be performed depending on the value of the MSB(s).

Algorithm 5 Ladder step executed for the first scalar bit equal to 1 ($k[i] = 1$)

Inputs: $x_P, X_1 = x_P, Z_1 = 1, X_2 = 1, Z_2 = 0$

1: $T_1 \leftarrow 1 + 0$
2: $X_2 \leftarrow 1 - 0$
3: $Z_2 \leftarrow x_P + 1$
4: $X_1 \leftarrow x_P - 1$
5: $T_1 \leftarrow 1 \cdot (x_P + 1)$
6: $X_2 \leftarrow 1 \cdot (x_P - 1)$
7: $Z_2 \leftarrow (x_P + 1) \cdot (x_P + 1) = (x_P + 1)^2$
8: $X_1 \leftarrow (x_P - 1) \cdot (x_P - 1) = (x_P - 1)^2$
9: $T_2 \leftarrow (x_P + 1)^2 - (x_P - 1)^2 = w_1$
10: $Z_1 \leftarrow w_1 \cdot a24 = w_2$
11: $Z_1 \leftarrow w_2 + (x_P - 1)^2 = w_3$
12: $Z_1 \leftarrow w_1 \cdot w_3 = w_4$
13: $X_1 \leftarrow (x_P + 1)^2 \cdot (x_P - 1)^2$
14: $Z_2 \leftarrow (x_P - 1) - (x_P + 1) = w_5$
15: $Z_2 \leftarrow w_5 \cdot w_5 = w_6$
16: $Z_2 \leftarrow w_6 \cdot x_P$
17: $X_2 \leftarrow (x_P - 1) + (x_P + 1) = w_7$
18: $X_2 \leftarrow w_7 \cdot w_7 = w_8$

return
$(X_1 = (x_P + 1)^2 \cdot (x_P - 1)^2, Z_1 = w_4);$
$(X_2 = w_8, Z_2 = w_6 x_P)$

Proposed countermeasure

Algorithm 6 Modified x-coordinate-based Montgomery Ladder

Inputs: $k \in \{0, ..., 2^{255} - 1\}, x_P$

$X_1 \leftarrow_{\$} \mathbb{F}_p$; $Z_1 \leftarrow_{\$} \mathbb{F}_p$; $X_2 \leftarrow_{\$} \mathbb{F}_p$; $Z_2 \leftarrow_{\$} \mathbb{F}_p$;
$W_1 \leftarrow x_P + 1$; $W_2 \leftarrow x_P - 1$; $p, s \leftarrow 0$;
for $i \leftarrow 254$ **downto** 0 **do**
  $t \leftarrow (k[i] \vee s) \oplus s$ ▷ $k[i]$ denotes bit i of k
  $c \leftarrow k[i] \oplus p$; $p \leftarrow k[i]$
  $(X_1, Z_1, X_2, Z_2) \leftarrow \mathsf{cswap}(X_1, Z_1, X_2, Z_2, c)$
  $(X_1, Z_1, X_2, Z_2) \leftarrow \mathsf{ladderstep}(x_p, X_1, Z_1, X_2, Z_2, W_1, W_2, t)$
  $s \leftarrow s \vee t$
end for
return (X_1, Z_1)

Initialise all variables with randomly generated, balanced, values

Iterate “dummy” loops for all MSBs=0

Execute special version of the loop for the first 1, with pre-calculated values, avoiding operations with variable = 1 or 0

Proposed countermeasure

Algorithm 7 Single Montgomery ladder step for the case $t = 1$

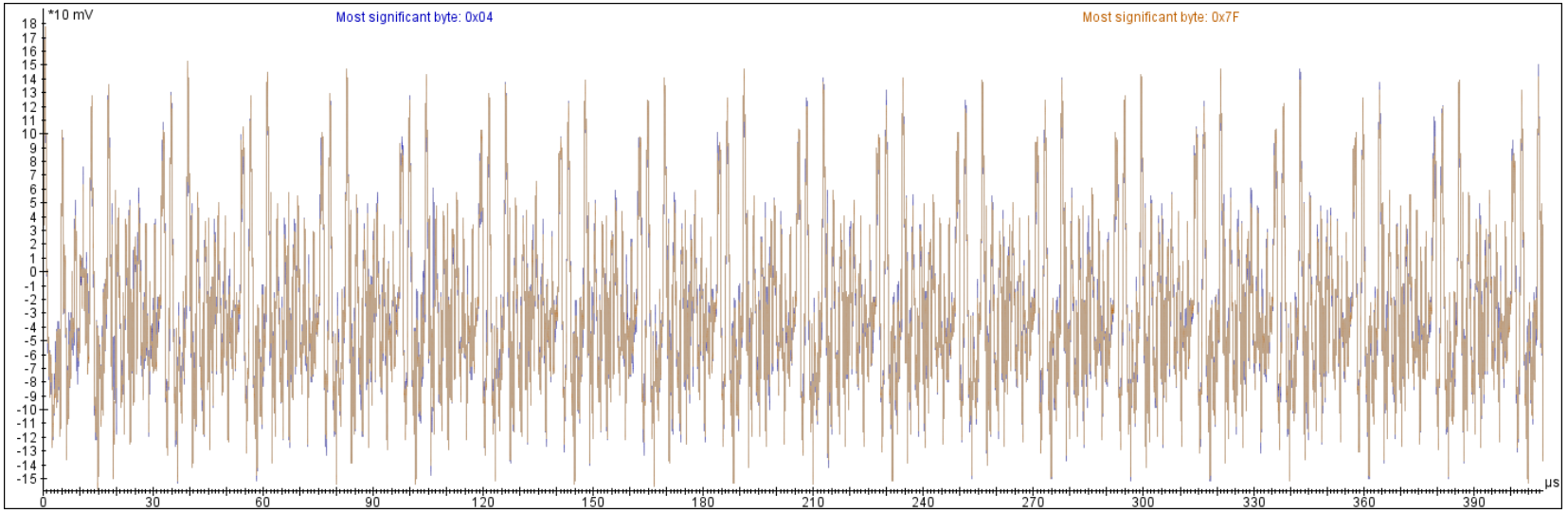
Inputs: $x_P, X_1, Z_1, X_2, Z_2, W_1, W_2$

1: $T_1 \leftarrow X_2 + Z_2$
2: $X_2 \leftarrow X_2 - Z_2$
3: $Z_2 \leftarrow X_1 + Z_1$
4: $X_1 \leftarrow X_1 - Z_1$
5: $T_1 \leftarrow T_1 \cdot X_1$
6: $X_2 \leftarrow X_2 \cdot Z_2$
7: $Z_2 \leftarrow W_1 \cdot W_1$
8: $X_1 \leftarrow W_2 \cdot W_2$
9: $T_2 \leftarrow Z_2 - X_1$
10: $Z_1 \leftarrow T_2 \cdot a24$
11: $Z_1 \leftarrow Z_1 + X_1$
12: $Z_1 \leftarrow T_2 \cdot Z_1$
13: $X_1 \leftarrow Z_2 \cdot X_1$
14: $Z_2 \leftarrow W_2 - W_1$
15: $Z_2 \leftarrow Z_2 \cdot Z_2$
16: $Z_2 \leftarrow Z_2 \cdot x_P$
17: $X_2 \leftarrow W_2 + W_1$
18: $X_2 \leftarrow X_2 \cdot X_2$

return (X_1, Z_1, X_2, Z_2)

Execute special version of the loop for the first 1, with pre-calculated values, avoiding operations with variables = 1 or 0

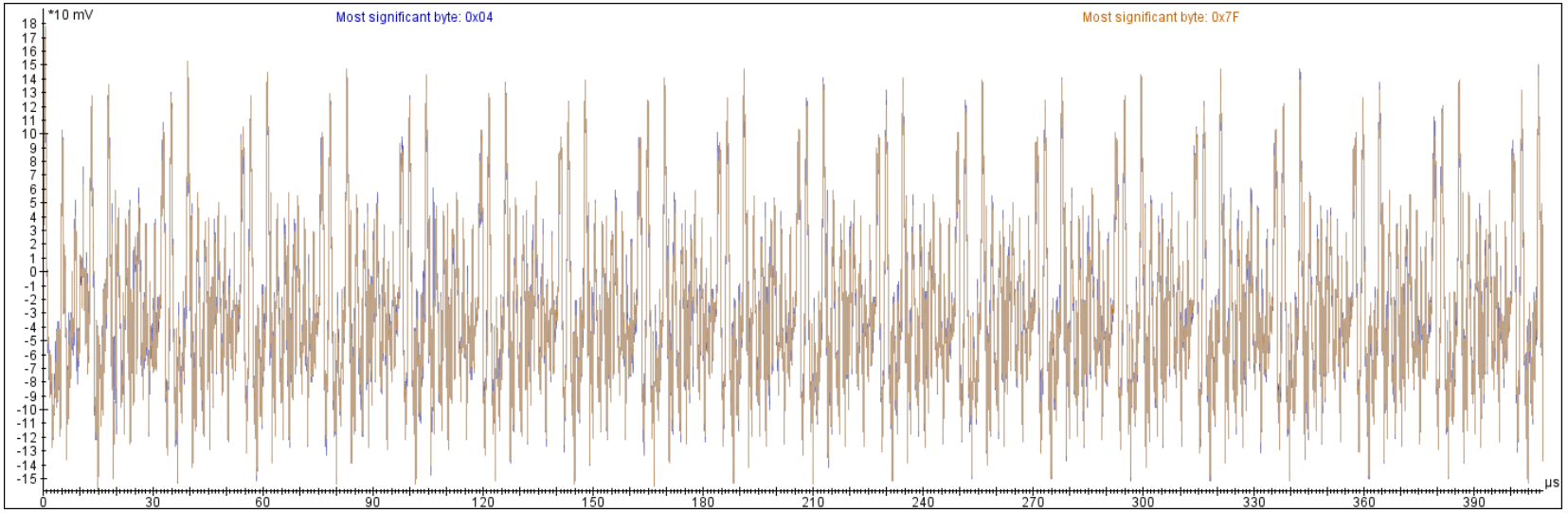
Proposed countermeasure



Swap the contents of two inputs if a value c=1; determine value c via arithmetic operations during each loop $\text{cswap}(X, Y, c)$

Alternative: place the content of the corresponding operands on different memory locations and access them according to values depending on the secret scalar

Proposed countermeasure



Implementation	Time (milliseconds):	Extra Memory (bytes):
Unprotected Imp.:	5.62	-
Cswap-based Imp.:	7.6 (+35.2%)	$8 * 32 = 256$
Secret-Memory Access Imp.:	5.81 (+3.4%)	$6 * 32 = 256$

Conclusions

We confirm that the MSBs of secret scalars can be easily extracted via SPA on open source implementations

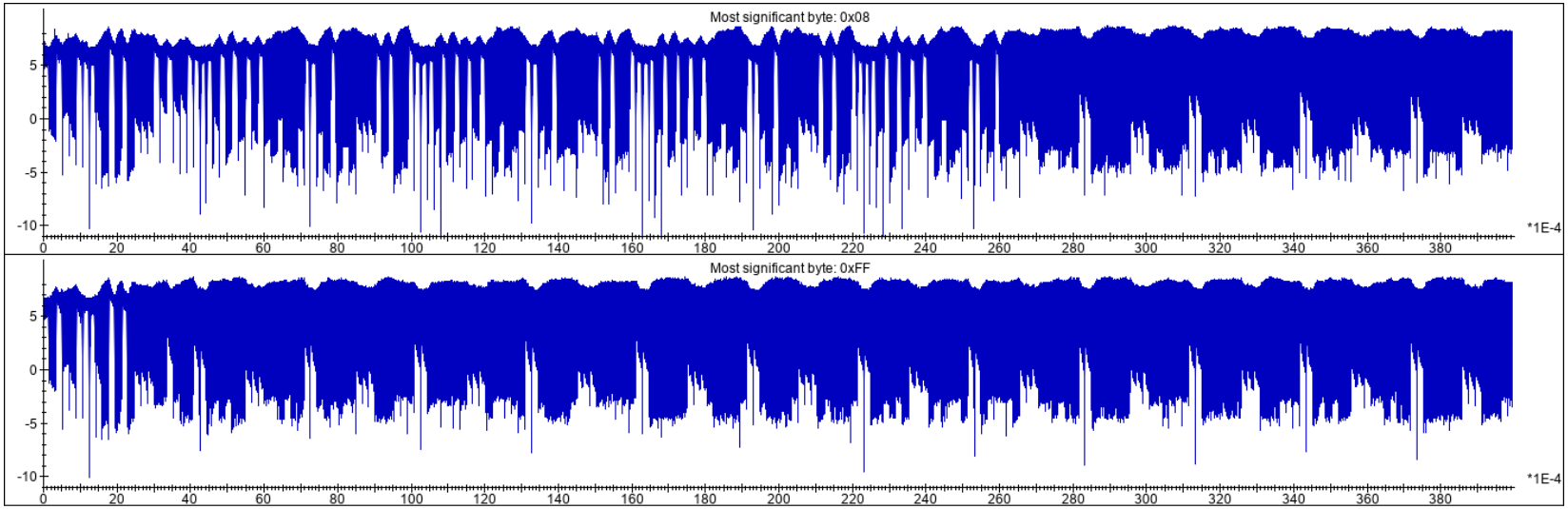
- The MSBs can be extracted on implementations of Cruve25519 and the Complete addition formulas

We propose a re-design of the algorithm implementing Curve25519 and show its effectiveness via power measurements

- We ensure that all loop iterations are performed using operands with long values
- Our re-design does not imply a big penalty on efficiency and size

We leave the proposal of a re-design mitigating this leakage on the complete addition formulas as future work

धन्यवाद



estuardo.alpirezbock@aalto.fi
Aalto University,
Department of Mathematics and Systems Analysis
Otakaari 1, Espoo
Room Y250c
Finland

Aalto University