Enhanced Elliptic Curve Scalar Multiplication Secure Against Side Channel Attacks and Safe Errors

Jeremy Dubeuf[1,2], David Hely[2], Vincent Beroulle[2]

(1): Maxim integrated, Security Excellence Lab - SEL

(2): University of Grenoble Alpes

Empowering Design Innovation

Agenda

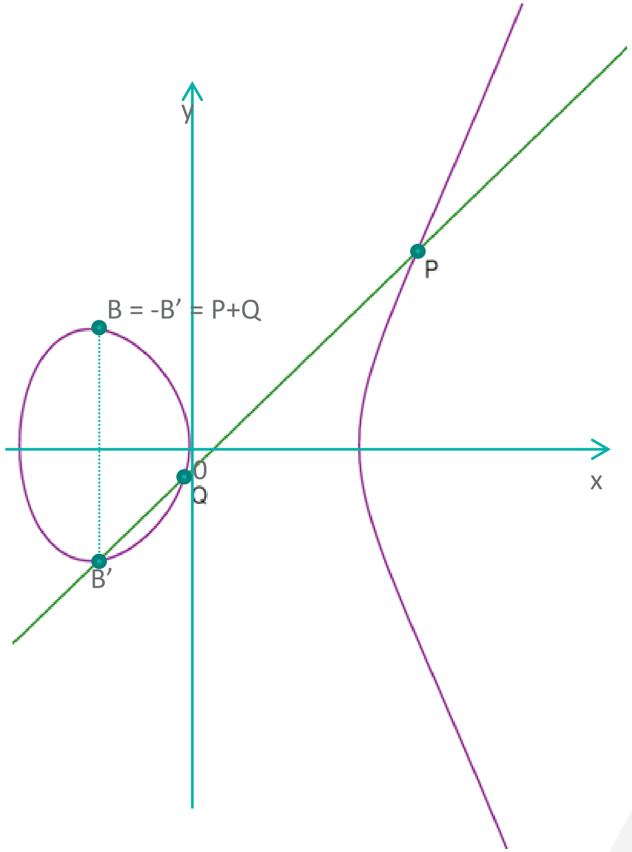
- Elliptic Curve cryptography and EC Scalar Multiplication
- Current common vulnerabilities of EC Scalar Multiplication algorithms
- A more Secure Scalar Multiplication algorithm
- Secure Implementation of Scalar Multiplication
- Conclusions

Elliptic Curve cryptography and Scalar Multiplication

ECC: Elliptic Curve Cryptography

Weierstrass equation: $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$

- A chord-and-tangent group law is defined between EC points.
- The scalar $k.P$ is defined by adding k time the point P
  - > $k.P = P + P + \ldots P$



P "+" Q = -B' = B

ECC: Elliptic Curve Cryptography

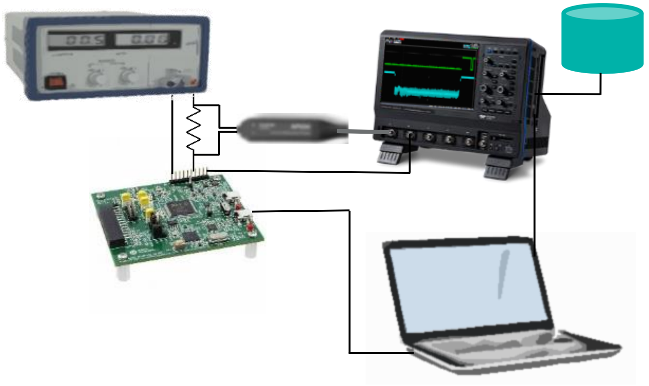
Different cryptographic schemes are build upon the scalar *k.P*

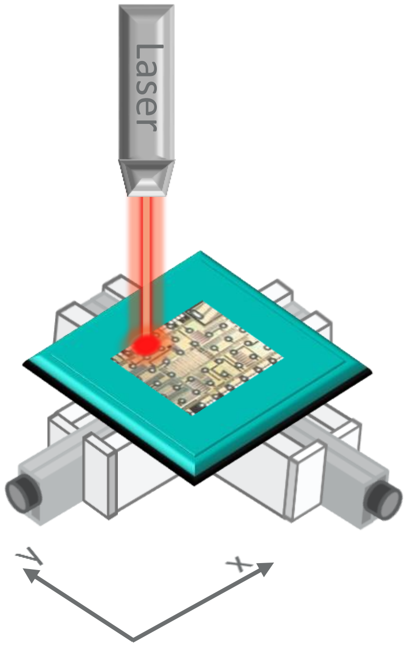
- ECDH: k is a secret key
- ECDSA: k is a random nonce during signature generation
- …
- This operation should be performed without leaking information on K

ECC: Elliptic Curve Cryptography

Physical implementations face-up a multitude of threats…

- **Side channel analysis** (power, EM, timing, noise…)
- **Faults** (power glitches, clock glitches, laser…)
- Reverse engineering
- µprobing
- …





Current common vulnerabilities of Scalar Multiplication algorithms

SPA: Simple Power Analysis

Algorithm 1 Left-to-right Double-and-Add

Input: $k = (k_{t-1}, k_{t-2}, \ldots, k_1, k_0)_2, P \in E(F_q)$

Output: kP

```
1: Q ← O
2: for i = t − 1 to 0 do
3:     Q ← 2Q
4:     if k_i then
5:         Q ← Q + P
6:     end if
7: end for
8: return (Q)
```

- In average, the execution time is : $\frac{t}{2}A + tD$
- The execution time vary between:

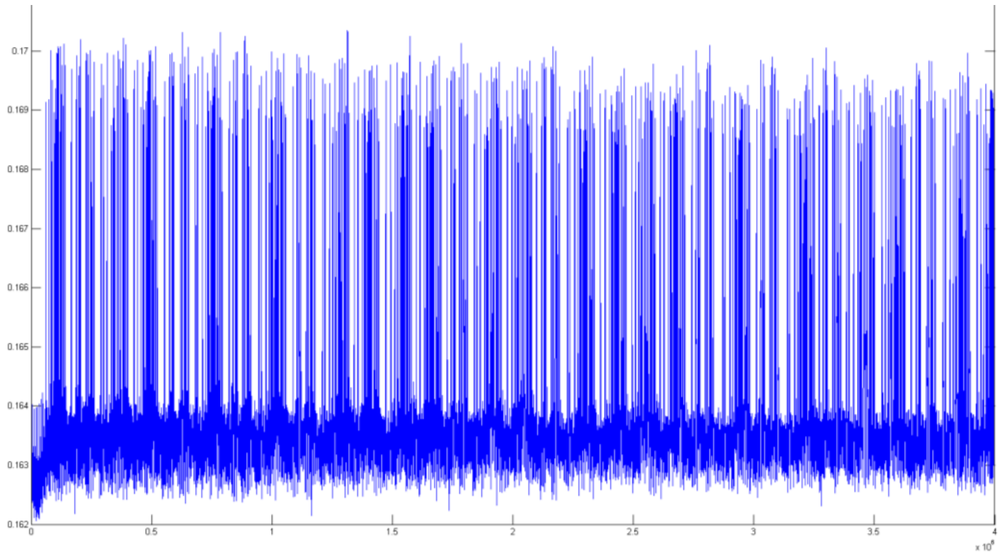
$$tD \leq T_{exec} \leq tA + tD$$

- If an attacker gets the global execution time, he gets the Hamming Weight.
- If an attacker gets each iteration time, he directly knows the key.

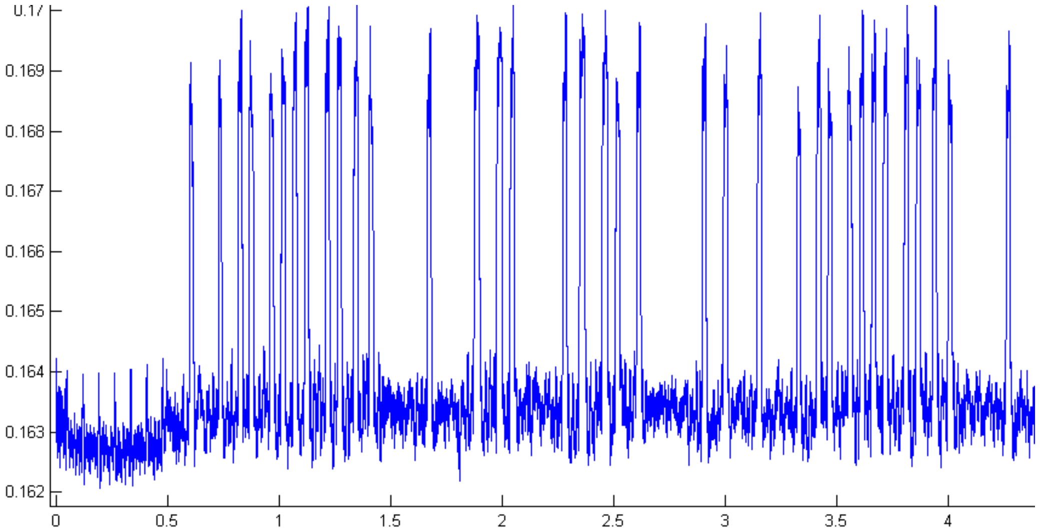
SPA: Simple Power Analysis

- An implementation of Alg. 1 provided this following power trace:

Overview of the power consumption :



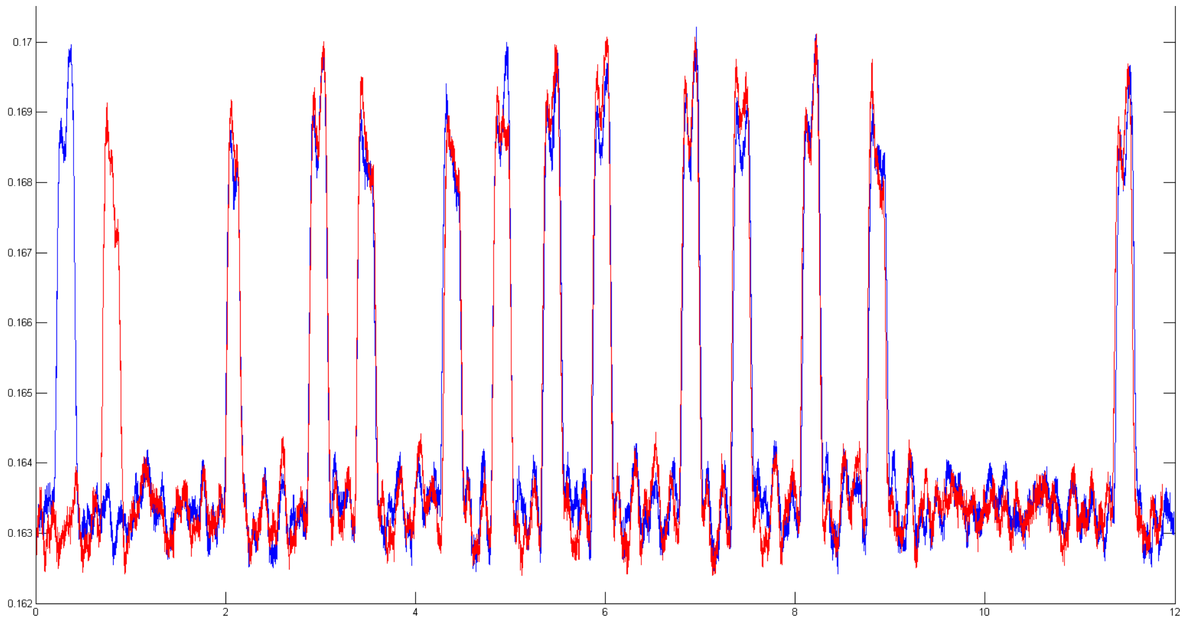
Zoom in:



maxim integrated™

SPA: Simple Power Analysis

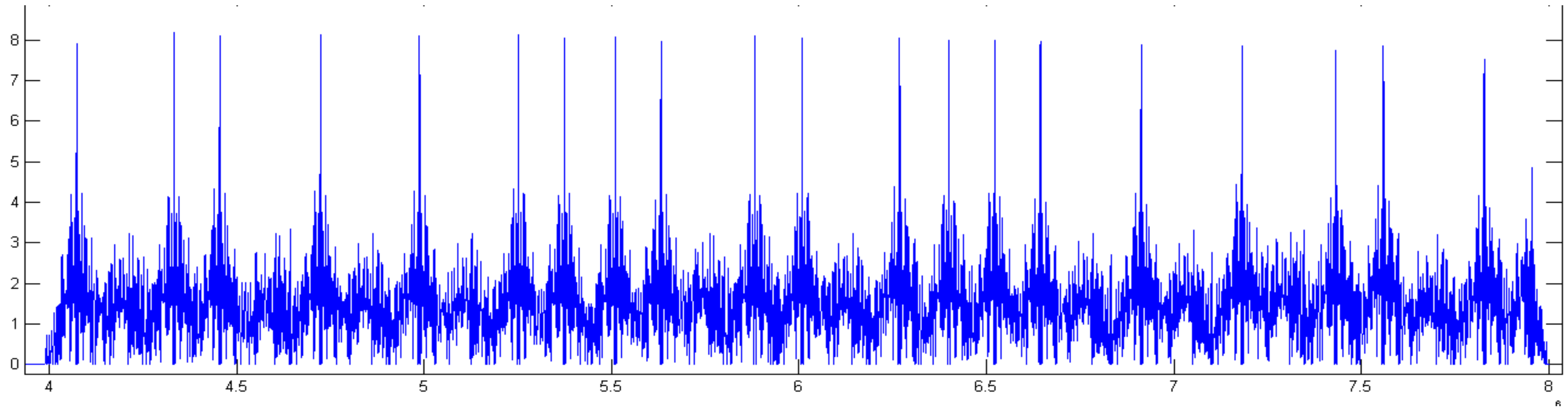
Two superposed patterns:



Note: -As this pattern is the first thing that appears in the power trace, it represents the point doubling operation

maxim integrated™

SPA: Simple Power Analysis



From the correlation we can thus said that this sequence is performed:

DA D DA DA DA D D D DA D DA D D D DA…

This corresponds to: 1 0 1 1 1 0 0 0 1 0 1 0 0 0 1… (DA=1, D=0)

The key K is 0xDC 51… = 1 1 0 1 1 1 0 0 0 1 0 1 0 0 0 1…

The first 1 is missed since no computation is done (data transfer)

Algorithm 1 Left-to-right Double-and-Add

Input: $k = (k_{t-1}, k_{t-2}, ..., k_1, k_0)_2, P \in E(F_q)$

Output: kP

```
Q ← O
for i = t − 1 to 0 do
    Q ← 2Q
    if k_i then
        Q ← Q + P
    end if
end for
return (Q)
```

SPA: Simple Power Analysis

As an SPA countermeasure, we can use an always add algorithm such Alg. 2:

Algorithm 2 Always Double-and-add

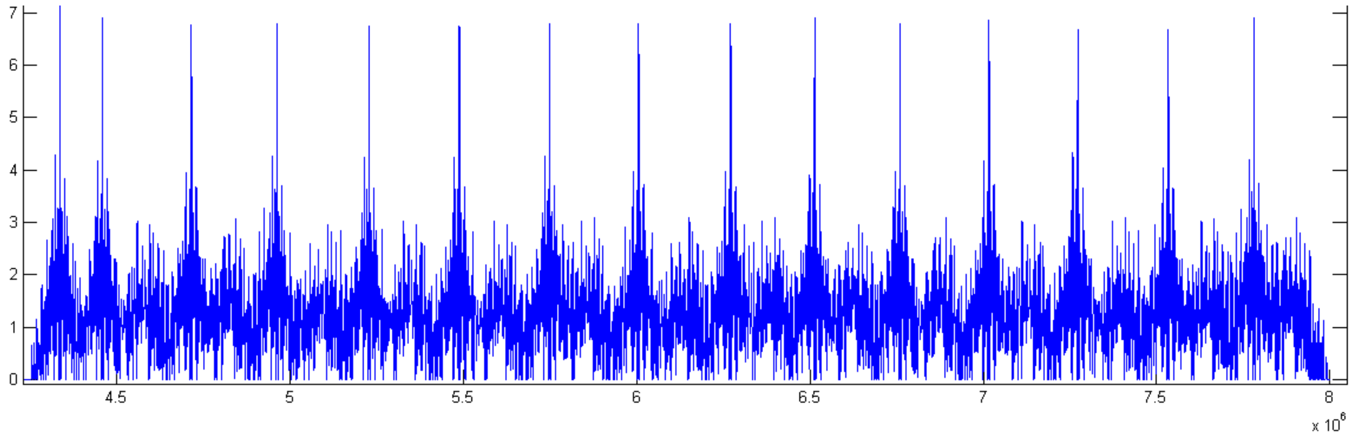
Input: $k = (k_{t-1}, ..., k_1, k_0)_2, P \in E(F_q)$
Output: $k.P$

$Q \leftarrow \infty$
for $i = t-1$ to 0 **do**
  $Q \leftarrow 2Q$
  if k_i **then**
    $Q \leftarrow Q + P$
  else
    $D \leftarrow Q + P$
  end if
end for
return (Q)

Always add algorithms have to be carefully implemented…

SPA: Simple Power Analysis

An always add algorithm was implemented, the correlation result is presented below :



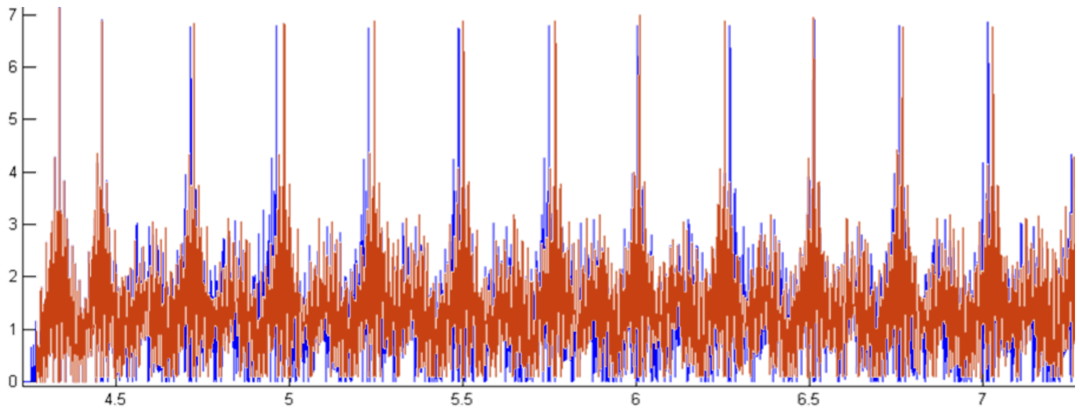
We can see a first small period that represent the first 1 value of the scalar. At the 1st one value, a data transfer occurs instead of an EC addition.

This leakage, due to the initialization $Q = \infty$, allows attackers to know the scalar length, thus the number of MSB bits set to 0.

SPA: Simple Power Analysis

Another leakage when implemented in software, is the use of the "if, else".

This structure generates a conditional branch that can lead to timing leakages due to cache hit/miss, wrong branch prediction, pipeline flushing etc.



SPA: MSB given away

- Alg. 3 avoids these two leakages:

Algorithm 3 Coron always Double-and-add

Input: $k = (1, k_{t-2}, \ldots, k_1, k_0)_2, P \in E(F_q)$

Output: $k.P$

```
Q[0] ← P
for i = t − 2 to 0 do
    Q[0] ← 2Q[0]
    Q[1] ← Q[0] + P
    Q[0] ← Q[k_i]
end for
return (Q[0])
```

Dummy operation!
=> C safe-error
Faults on Q[0]+P computation or on Q[1] reveal information

- It uses a two-indexes table to avoid the “if, else” condition.
- The infinity point is avoided thanks to the initialization.

=> A naive implementation of this algorithm either results in having a loop dependent of the length of the secret scalar (i.e. reduced to the first non-null bit) or to give away 1 bit by forcing the MSB.

Faults: Local dummy operation
C safe-error against Montgomery ladder

The Montgomery Ladder is often presented as a C safe-error resistant algorithm:

Algorithm 4 Montgomery scalar operation

Input: $k = (1, k_{t-2}, ..., k_1, k_0)_2, P \in E(F_q)$
Output: $k.P$

$R_0 \leftarrow P$
$R_1 \leftarrow 2P$
for $i = t - 2$ to 0 **do**
  $R_{1-k_i} \leftarrow R_0 + R_1$
  $R_{k_i} \leftarrow 2R_{k_i}$
end for
return (R_0)

However:

- The scalar MSB should be 1

Faults: Local dummy operation
C safe-error against Montgomery ladder

The Montgomery Ladder is often presented as a C safe-error resistant algorithm:

Algorithm 4 Montgomery scalar operation

Input: $k = (1, k_{t-2}, ..., k_1, k_0)_2, P \in E(F_q)$
Output: $k.P$

$R_0 \leftarrow P$
$R_1 \leftarrow 2P$
for $i = t - 2$ to 0 **do**
    $R_{1-k_i} \leftarrow R_0 + R_1$
    $R_{k_i} \leftarrow 2R_{k_i}$
end for
return (R_0)

In k = 0 case

Algorithm 4 Montgomery scalar operation

Input: $k = (1, 0, ..., 0, 0)_2, P \in E(F_q)$
Output: $k.P$

$R_0 \leftarrow P$
$R_1 \leftarrow 2P$
for $i = t - 2$ to 0 **do**
    $R_1 \leftarrow R_0 + R_1$
    $R_0 \leftarrow 2R_0$
end for
return (R_0)

However:

- The scalar MSB should be 1
- For k = 0 the operation $R_{1-k_i} \leftarrow R_0 + R_1$ becomes a dummy operation.

Faults: Local dummy operation
C safe-error against Montgomery ladder

The Montgomery Ladder is often presented as a C safe-error resistant algorithm:

Algorithm 4 Montgomery scalar operation

Input: $k = (1, k_{t-2}, ..., k_1, k_0)_2, P \in E(F_q)$
Output: $k.P$

1: $R_0 \leftarrow P$
2: $R_1 \leftarrow 2P$
3: **for** $i = t - 2$ to 0 **do**
4: $\quad R_{1-k_i} \leftarrow R_0 + R_1$
5: $\quad R_{k_i} \leftarrow 2R_{k_i}$
6: **end for**
7: **return** (R_0)

Transient faults allow to attack any bit

However:

- The scalar MSB should be 1
- For k = 0 the operation $R_{1-k_i} \leftarrow R_0 + R_1$ becomes a dummy operation and R_1 is never used.

=> An attacker may inject faults into $R_{1-k_i} \leftarrow R_0 + R_1$ for some k LSBs and then deduce if k LSBs = 0

=> Vulnerable to C safe-error!

Faults: Unused memory values

The BRIP algorithm aims at using a random point to thwart CPA and other data dependent attacks.

Algorithm 5 Binary Expansion with RIP (BRIP)

Input: $k = (k_{t-1}, ..., k_1, k_0)_2, P \in E(F_q)$
Output: $k.P$

$R \leftarrow randompoint()$
$T \leftarrow P - R$
$Q \leftarrow R$
for $i = t - 1$ to 0 **do**
    $Q \leftarrow 2Q$
    if k_i **then**
        $Q \leftarrow Q + T$
    else
        $Q \leftarrow Q - R$
    end if
end for
return $(Q - R)$

Transient faults allow to attack any bit

However:

- For k = 0, T is never involved in the result!

=> An attacker may fault T prior the evaluation of some k LSBs and then deduce if k LSBs = 0

Infinity point and dummy operands

Edward curves are often promoted for its unified and complete addition law

$$(x_1, y_1) + (x_2, y_2) = (\frac{x_1 y_2 + x_2 y_1}{1 + d x_1 x_2 y_1 y_2}, \frac{y_1 y_2 + x_1 x_2}{1 - d x_1 x_2 y_1 y_2})$$

However with the neutral element, it becomes:

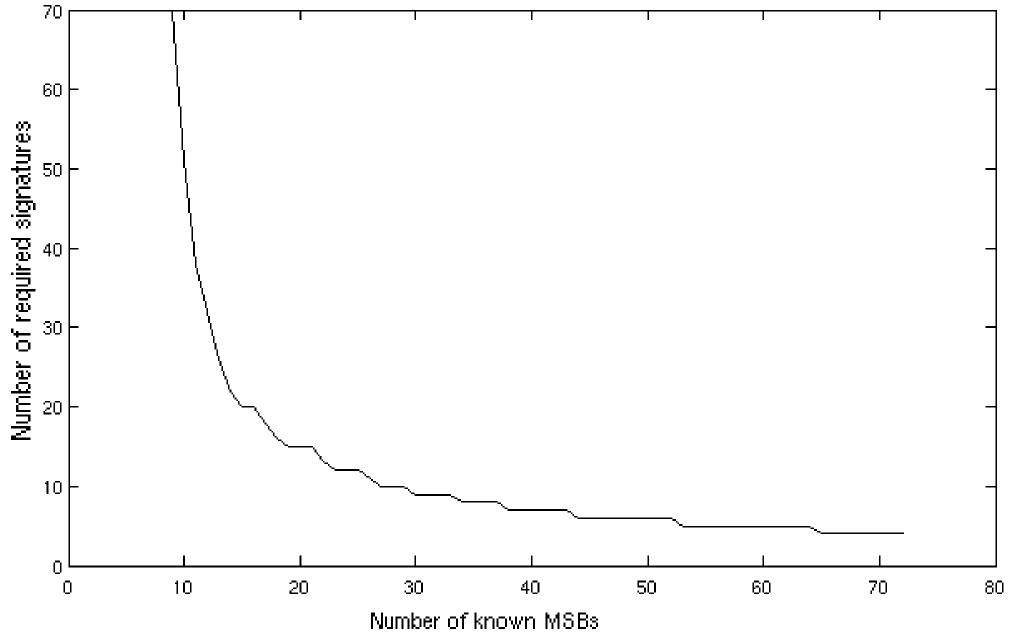
$$(x_1, y_1) + (0, 1) = (\frac{x_1 \cdot 1 + 0 \cdot y_1}{1 + d x_1 \cdot 0 \cdot y_1 \cdot 1}, \frac{y_1 \cdot 1 + x_1 \cdot 0}{1 - d x_1 \cdot 0 \cdot y_1 \cdot 1}) = (x_1, y_1)$$

Does this formula is really “unified” from a power/EM point of view?

What happen if x_1 or y_1 is faulted with the “good “timing?

Lattice & Bleinchenbacher attacks against ECDSA

Lattice against NIST P256:



- Only 70 signatures with 9 known bits are enough!
- This is just a basic implementation…

Bleichenbacher:

q	160 bits			192 bits	256 bits
ℓ	1	2	2	2	3
m	2^{26}	2^{14}	200	2^{16}	2^{16}
Tech.	Bleich.	Bleich.	Latt.	Bleich.	Bleich.
Compl.	2^{40}	2^{28}	Few hr	2^{33}	2^{33}

Figure from: Fouque, P.A., Guilley, S., Murdica, C., Naccache, D.: Safe-Errors on SPA Protected implementations with the Atomicity Technique. IACR Cryptology ePrint Archive 2015, 794 (2015)

- **A few leaked bits is enough…**

Lessons learnt

We should:

- Avoid operation flow dependency from the scalar and keep it constant (side channel + CFI).
- Avoid any dummy operation or operand, even locally (fault).
- Avoid the infinity point (side channel + fault).
- Avoid constraints on the scalar (MSB, even, odd…).
- Avoid any specific scalar representation to avoid leakages on the transformation function.
- Avoid HW leakage to preserve entropy.

A more Secure Scalar Multiplication algorithm

Scalar algorithm basic

Algorithm 6 scalar operation

Input: $E(F_q), k = (k_{t-1}, ..., k_1, k_0)_2, P \in E(F_q)$
Output: $2k.P$

$Q \leftarrow P$
for $i = t - 1$ to 0 **do**
    $Q \leftarrow 2.Q$
    $Q \leftarrow Q + (-1)^{\overline{k_i}}.P$ //add or subtract P, depending on k_i
end for
$Q \leftarrow Q - P$
return (Q)

- Constant
- Dummy operation free
- No unused memory values
- No infinity point
- Scalar constraints free

This algorithm does not prevent data dependent leakage
Also note that if MSBs = 0 then the for loop computes: Q←2P, Q←(2P)-P, Q←2P…

The classic random projective coordinates countermeasure corrects both issues

Jacobian-affine P±Q

Algorithm 7 ECC Jacobian-affine point addition/subtraction

Input: $P = (X_1 : Y_1 : Z_1)$ in Jacobian and $Q = (x_2, y_2)$ in affine $\in E(F_q)$, b operation selection, r a random bit
Output: if $b = 0$, $P - Q$ else $P + Q$

```
T_1 ← Z_1^2
T_2 ← T_1 · Z_1
T_1 ← T_1 · x_2
T_3[r] ← −T_2
T_3[r̄] ← T_2
T_2 ← T_3[b ⊕ r] · y_2
T_1 ← T_1 − X_1
T_2 ← T_2 + T_3[0]
T_2 ← T_2 + T_3[1]
T_2 ← T_2 − Y_1
if T_1 == 0 then
    return (error)
end if
Z_3 ← Z_1 · T_1
T_3[0] ← T_1^2
T_3[1] ← T_3[0] · T_1
T_3[0] ← T_3[0] · X_1
T_1 ← T_3[0] + T_3[0]
X_3 ← T_2^2
X_3 ← X_3 − T_1
X_3 ← X_3 − T_3[1]
T_3[0] ← T_3[0] − X_3
T_3[0] ← T_3[0] · T_2
T_3[1] ← T_3[1] · Y_1
Y_3 ← T_3[0] − T_3[1]
```

- A two-indexes table is used, b selects the + or – operation
- Both indexes are used in order to propagate fault to the result independently of the faulted index

Scalar algorithm – Comb compliant

Algorithm 8 Modified Comb method

Input: $E(F_q), k = (k_{t-1}, ..., k_1, k_0)_2, P \in E(F_q)$, window width $w, d = \lceil \frac{t}{w} \rceil$

Output: $2k.P$

Pre-computation: compute $2 \cdot [1, a_{w-2}, ...a_1, a_0].P - [1_{w-1}, ...1_1, 1_0].P$ for all possible binary values of $a_{w-2}, ...a_1, a_0$, with $[1, a_{w-2}, ...a_1, a_0].P = 2^{(w-1)d}P + ... + a_1 2^d P + a_0 P$.

Represent k as: $\begin{pmatrix} k_{d-1}^0 & \cdots & k_1^0 & k_0^0 \\ \vdots & \ddots & \ddots & \vdots \\ k_{d-1}^{w-1} & \cdots & k_1^{w-1} & k_0^{w-1} \end{pmatrix}$ //if necessary, pad 0s as k MSBs.

$Q \leftarrow [1_{w-1}, ...1_1, 1_0].P$ //represents the highest pre-calculated point.
for $i = d - 1$ to 0 **do**
  $Q \leftarrow 2.Q$
  $Q \leftarrow Q + (-1)^{\overline{k_i^{w-1}}}.\overline{[k_i^{w-1} \oplus [k_i^{w-2}, ...k_i^1, k_i^0]]}.P$
end for
$Q \leftarrow Q - [1_{w-1}, ...1_1, 1_0].P$
return (Q)

This algorithm does not use all pre-computed point for each loop iteration…

Secure Implementation of Scalar Multiplication

Scalar algorithm – Different use cases

Depending on the cryptographic scheme, either:

- k.P with P a constant base point
- k.P + v.G, with G coming from outside the system
- k.G, with G coming from outside the system

➡ Can we come up with a general implementation for all use cases?

Scalar algorithm – k.P

Algorithm 9 kP operation

Input: $k = (k_{t-1}, ..., k_1, k_0)_2$, P and $2^{t/2}P \in E(F_q)$
Output: $2k.P$

$r \leftarrow randombit()$
$P[r] \leftarrow 2^{t/2}P - P$ //in Affine coordinates
$P[\overline{r}] \leftarrow 2^{t/2}P + P$ //in Affine coordinates
$Q \leftarrow RandomAfftoJac(P[1])$ //use random affine to Jacobian conversion
for $i = t/2 - 1$ to 0 **do**
    $Q \leftarrow 2Q$
    $Q \leftarrow Q + (-1)^{\overline{k_{i+t/2}}} P[r \oplus \overline{(k_i \oplus k_{i+t/2})}]$
    $r \leftarrow randombit()$ //refresh the random r
    $shuffleregisters(P[0], P[1], r)$ //shuffle P[0] and P[1] according to r
end for
$Q \leftarrow Q + P[r]$ //add $P[r]$ for system integrity
$Q \leftarrow Q - P[\overline{r}]$
$Q \leftarrow Q - P[r]$ //remove $P[r]$
$Q \leftarrow JactoAff(Q)$ //use Jacobian to affine conversion
$Verify(Q)$
return Q

- Comb with 2 points
- $\frac{t}{2} D + (\frac{t}{2} + 5) A$

Scalar algorithm – k.P + v.G

Algorithm 10 $k \cdot P + v \cdot G$ operation

Input: $k = (k_{t-1}, ..., k_1, k_0)_2$, $v = (v_{t-1}, ..., v_1, v_0)_2$, P and $G \in E(F_q)$
Output: $2kP + 2vG$

$r \leftarrow$ *randombit()*
$P[r] \leftarrow G - P$ //in Affine coordinates
$P[\overline{r}] \leftarrow G + P$ //in Affine coordinates
$Q \leftarrow$ *RandomAfftoJac*$(P[1])$ //use random affine to Jacobian conversion
for $i = t - 1$ to 0 **do**
    $Q \leftarrow 2Q$
    $Q \leftarrow Q + (-1)^{\overline{v_i}} P[r \oplus \overline{(k_i \oplus v_i)}]$
    $r \leftarrow$ *randombit()* //refresh the random r
    shuffleregisters$(P[0], P[1], r)$ //shuffle P[0] and P[1] according to r
end for
$Q \leftarrow Q + P[r]$ //add $P[r]$ for system integrity
$Q \leftarrow Q - P[\overline{r}]$
$Q \leftarrow Q - P[r]$ //remove $P[r]$
$Q \leftarrow$ *JactoAff*(Q) //use Jacobian to affine conversion
Verify(Q)
return Q

Scalar algorithm – k.P vs kP+v.G

Algorithm 9 kP operation

Input: $k = (k_{t-1}, ..., k_1, k_0)_2$, P and $2^{t/2}P \in E$

Output: $2k.P$

```
r ← randombit()
P[r] ← 2^{t/2}P − P                          //in Af
P[r̄] ← 2^{t/2}P + P                          //in Af
Q ← RandomAfftoJac(P[1])                      //use ra
for i = t/2 − 1 to 0 do
    Q ← 2Q
    Q ← Q + (−1)^{k_{i+t/2}} P[r ⊕ (k_i ⊕ k_{i+t/2})‾]
    r ← randombit()                            //ref
    shuffleregisters(P[0],P[1],r)              //shuffle
end for
Q ← Q + P[r]                                   //ad
Q ← Q − P[r̄]
Q ← Q − P[r]                                   //rei
Q ← JactoAff(Q)                                //use .
Verify(Q)
return Q
```

Algorithm 10 $k \cdot P + v \cdot G$ operation

Input: $k = (k_{t-1}, ..., k_1, k_0)_2$,$v = (v_{t-1}, ..., v_1, v_0)_2$, P and G

Output: $2kP + 2vG$

```
r ← randombit()
P[r] ← G − P                                   //in Affine coordinates
P[r̄] ← G + P                                   //in Affine coordinates
Q ← RandomAfftoJac(P[1])                        //use random affine
for i = t − 1 to 0 do
    Q ← 2Q
    Q ← Q + (−1)^{v̄_i} P[r ⊕ (k_i ⊕ v_i)‾]
    r ← randombit()                              //refresh the ranc
    shuffleregisters(P[0],P[1],r)                //shuffle P[0] and P[1
end for
Q ← Q + P[r]                                     //add P[r] for sys
Q ← Q − P[r̄]
Q ← Q − P[r]                                     //remove P[r]
Q ← JactoAff(Q)                                  //use Jacobian to a
Verify(Q)
return Q
```

Scalar algorithm – k.G

Algorithm 11 kG operation computed as $k \cdot G = k_1 G + k_2(r \cdot G)$

Input: $k = (k_{t-1}, ..., k_1, k_0)_2$, $G \in E(F_q)$
Output: $2k \cdot G$

1: $r \leftarrow random([0, 2^{32}])$
2: $v \leftarrow random([0, \#E(F_q) - 1])$
3: $k \leftarrow k - v$
4: $v \leftarrow v \cdot r^{-1} \mod n$
5: $Q \leftarrow AlgKP(r, G)$ //Algorithm 9 reduced for 32bits of scalar
6: $R \leftarrow AlgkPvG(k, G, v, Q)$ //Algorithm 10 kP+vQ
7: **return** R

- $kP = k_1P + k_2(rP)$, move from t bits of secret to $2t + 32$
- k_1 is used for precomputed point selection, k_2 is used for add/sub selection.
- r is used to reduce the statistical dependency between k_1 and k_2 bits.
- Only ~12% slower than the classic double-and-add always for NIST P256, can be adjusted depending on r.

Conclusions

- A couple of bits are enough to defeat an ECDSA scheme.
- Grab a couple of bits thanks to safe-error or side channel is easy in most currently available EC scalar algorithms.
- Permanent and temporary faults should be considered.
- Basic scalar blinding does not mask all the scalar…
- Both, operation flow and data can be targeted by a fault.

- We provided an algorithm secure against both side channel and safe-error with performance slightly slower (12%) than a basic double-and-add always.
- Transient and permanent faults were considered.

Thank You!