Randomized yet constant-time elliptic curve scalar multiplication

Laurent Imbert

ECO research group, LIRMM, Montpellier

Ongoing joint work with

Jérémie Detrey (CARAMBA, LORIA)

RAIM, Lyon

October 2017

Elliptic curve crypto in a nutshell

- An elliptic curve E is a finite abelian group whose elements are points in a projective space.
- Group law is denoted additively:
  - addition: $P, Q \longmapsto P + Q$
  - doubling: $P \longmapsto [2]P$
- Neutral element is denoted $\mathcal{O}$.
- $P + (-P) = \mathcal{O}$
- Curve arithmetic: evaluation of multivariate polynomials
- Scalar multiplication:

$$k \in \mathbb{Z},\ P \in E \longmapsto [k]P = P + P + \cdots + P$$

Elliptic curve crypto in a nutshell (cont.)

- Crypto requires elliptic curves defined over finite fields.
- The order ℓ of an elliptic curve defined over $\mathbb{F}_q$ is finite.
- Hasse's theorem:

$$|\ell - (q+1)| \leq 2\sqrt{q}$$

Elliptic curve crypto in a nutshell (cont.)

- Crypto requires elliptic curves defined over finite fields.
- The order ℓ of an elliptic curve defined over $\mathbb{F}_q$ is finite.
- Hasse's theorem:
$$|\ell - (q+1)| \leq 2\sqrt{q}$$
- Discrete logarithm problem in E: given P and $Q = [k]P$, find k.
- Complexity of the best known ECDL algorithm: $O(\sqrt{\ell})$.
- Key sizes: a 256-bit group provides 128 bits of security.

Breaking a cryptosystem

Solving the underlying mathematical problem (integer factorization, discrete log., lattice reduction problems, etc.) should be unfeasible.

Breaking a cryptosystem

Solving the underlying mathematical problem (integer factorization, discrete log., lattice reduction problems, etc.) should be unfeasible.

Side Channel Attacks (SCA) [Kocher'96]

- Integrated circuits leak information (computation time, power consumption, electromagnetic emanations, etc.).
- Amount of leakage depends on transistor activity, itself closely related to the performed operations and processed data.
- When a device is processing secret values, data-dependent leakage can expose these secrets to attacks.
- Both private and public-key algorithms are vulnerable.

The simplest attack

- SPA, SEMA only consider the performed operations.
- They only require one execution of the algorithm.
- A single trace may be sufficient to reveal the whole secret key!

The simplest attack

- SPA, SEMA only consider the performed operations.
- They only require one execution of the algorithm.
- A single trace may be sufficient to reveal the whole secret key!

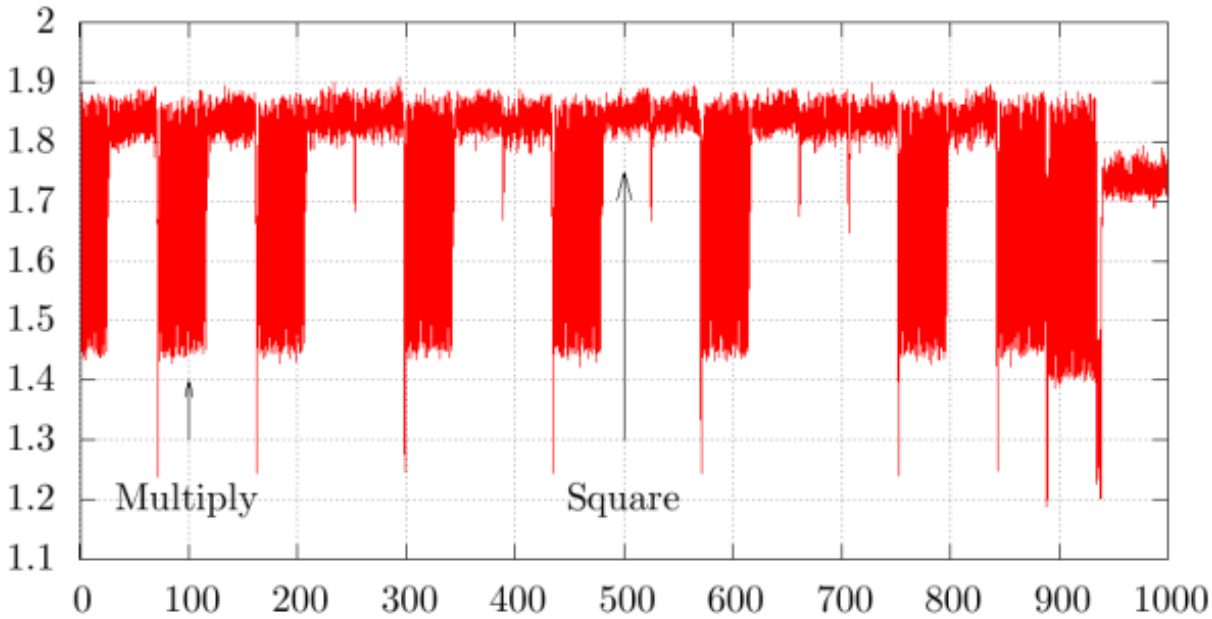

Fig.: O. Meynard et. al. Enhancement of Simple Electro-Magnetic Attacks by Pre-characterization in Frequency Domain and Demodulation Techniques. DATE 2011

Equipment

Photo Victor Lomné - NinjaLab - Montpellier

Equipment

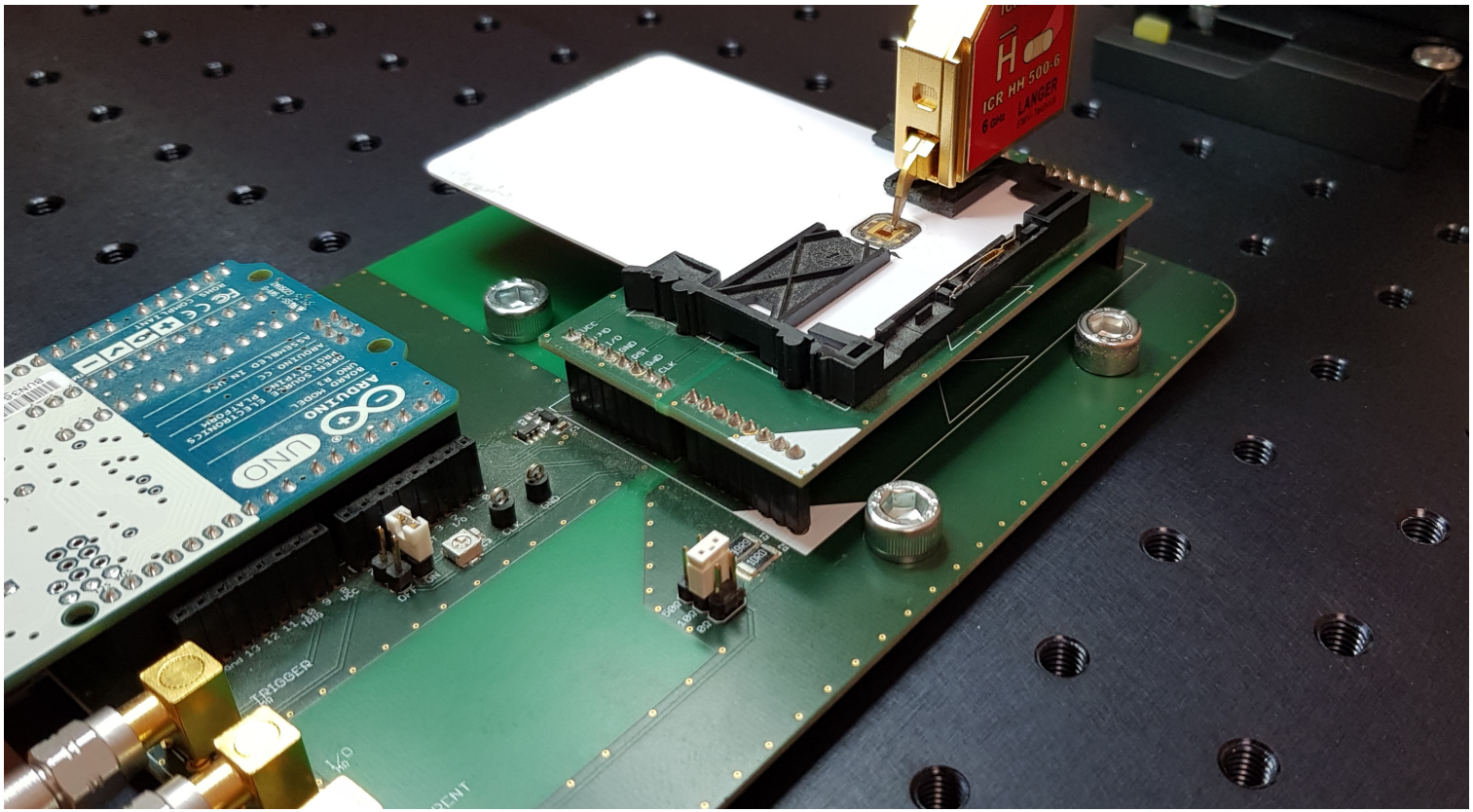

Photo Victor Lomné - NinjaLab - Montpellier

Game rules for designing robust algorithms

- **Simple attacks**
  Sequence of operations should be independent from the secret
- **Differential/Template attacks**
  Randomization strategies (scalar, point, coordinates, algorithm)
- **Timing attacks**
  Constant-time algorithms
- **Cache-timing attacks**
  No memory access at secret-data-dependent addresses
- **Fault attacks**
  No dummy operations

Randomized Algorithms

Addition-Subtraction chain for computing k

A sequence of integers $(a_0 = 1, a_1, a_2, \ldots, a_{n-1}, a_n = k)$ such that for all $r > 0$, $a_r = a_s \pm a_t$ for some $0 \leq s, t < r$

Such a chain (of length $n + 1$) for computing k provides a scalar multiplication algorithm for computing $[k]P$ which requires n point operations (additions/subtractions/doublings).

In general, there are very many chains for computing k.

Randomized Algorithms

Addition-Subtraction chain for computing k

A sequence of integers $(a_0 = 1, a_1, a_2, \ldots, a_{n-1}, a_n = k)$ such that for all $r > 0$, $a_r = a_s \pm a_t$ for some $0 \leq s, t < r$

Such a chain (of length $n + 1$) for computing k provides a scalar multiplication algorithm for computing $[k]P$ which requires n point operations (additions/subtractions/doublings).

In general, there are very many chains for computing k.

Proposed countermeasure against differential attacks

Use a different, randomly selected addition-subtraction chain for each scalar multiplication

Binary signed digits failures [Oswald, Aigner'01], [Ha, Moon'02].

Let's play!

Binary algorithm (aka double-and-add)

$$[k]P = \begin{cases} [2]([k/2]P) & k \text{ even} \\ [2]([(k-1)/2]P) + P & k \text{ odd} \end{cases}$$

Let's play!

Binary algorithm (aka double-and-add)

$$[k]P = \begin{cases} [2]([k/2]P) & k \text{ even} \\ [2]([(k-1)/2]P) + P & k \text{ odd} \end{cases}$$

$$[k]P = [2]Q + [r]P, \quad \text{where } Q = [k/2]P, \quad r = k \bmod 2$$

Let's play!

Binary algorithm (aka double-and-add)

$$[k]P = \begin{cases} [2]([k/2]P) & k \text{ even} \\ [2]([(k-1)/2]P) + P & k \text{ odd} \end{cases}$$

$$[k]P = [2]Q + [r]P, \quad \text{where } Q = [k/2]P, \quad r = k \bmod 2$$

m-ary algorithm

$$[k]P = [m]Q + [r]P, \quad \text{where } Q = [k/m]P, \quad r = k \bmod m$$

Playing with multiple moduli

Let $\mathcal{M} = \{m_1, m_2, \ldots, m_n\}$ a predefined set of moduli

Playing with multiple moduli

Let $\mathcal{M} = \{m_1, m_2, \ldots, m_n\}$ a predefined set of moduli

Pick m_{i_0} at random in $\mathcal{M}$

$[k]P = [m_{i_0}]Q_0 + [r_{i_0}]P$

Compute $Q_0 = [k/m_{i_0}]P$

Playing with multiple moduli

Let $\mathcal{M} = \{m_1, m_2, \ldots, m_n\}$ a predefined set of moduli

Pick m_{i_0} at random in $\mathcal{M}$

$[k]P = [m_{i_0}]Q_0 + [r_{i_0}]P$

Compute $Q_0 = [k/m_{i_0}]P$

Pick m_{i_1} at random in $\mathcal{M}$

$[k/m_{i_0}]P = [m_{i_1}]Q_1 + [r_{i_1}]P$

Compute $Q_1 = [(k/m_{i_0})/m_{i_1}]P$

$\ldots$

A recursive randomized scalar multiplication

Input: $\mathcal{M} = \{m_1, \ldots, m_n\}$, $k \in \mathbb{N}, P \in E$

Output: $Q = [k]P \in E$

if $k = 0$ **then**

return $\mathcal{O}$

else if $k = 1$ **then**

return P

Pick m at random in $\mathcal{M}$

compute $r \leftarrow k \bmod m$

compute $Q \leftarrow [(k - r)/m]P$ recursively

return $[m]Q + [r]P$

A recursive randomized scalar multiplication

Input: $\mathcal{M} = \{m_1, \ldots, m_n\}$, $k \in \mathbb{N}, P \in E$

Output: $Q = [k]P \in E$

if $k = 0$ **then**

return $\mathcal{O}$

else if $k = 1$ **then**

return P

Pick m at random in $\mathcal{M}$

compute $r \leftarrow k \bmod m$

compute $Q \leftarrow [(k - r)/m]P$ recursively

return $[m]Q + [r]P$

$$[k]P = [r_{i_0}]P + [m_{i_0}]\left([r_{i_1}]P + [m_{i_1}]\left([r_{i_2}]P + \cdots + [m_{i_{s-1}}]([r_{i_s}]P)\cdots\right)\right)$$

Mixed-radix scalar multiplication

MRS representation of k in base $(m_{i_0}, m_{i_1}, \ldots, m_{i_s})$:

$$k = r_{i_0} + m_{i_0}\left(r_{i_1} + m_{i_1}\left(r_{i_2} + \cdots + m_{i_{s-1}}(r_{i_s} + m_{i_s}.0)\cdots\right)\right)$$

Mixed-radix scalar multiplication

MRS representation of k in base $(m_{i_0}, m_{i_1}, \ldots, m_{i_s})$:

$$k = r_{i_0} + m_{i_0}\left(r_{i_1} + m_{i_1}\left(r_{i_2} + \cdots + m_{i_{s-1}}(r_{i_s} + m_{i_s}.0)\cdots\right)\right)$$

An iterative version

Input: $\mathcal{M} = \{m_1, \ldots, m_n\}$, $k \in \mathbb{N}$, $P \in E$

Output: $Q = [k]P \in E$

compute a randomized MRS representation of k

$Q \leftarrow \mathcal{O}$

for $j = s$ downto 0 **do**

$Q \leftarrow [m_{i_j}]Q + [r_{i_j}]P$

return Q

Mixed-radix scalar multiplication

MRS representation of k in base $(m_{i_0}, m_{i_1}, \ldots, m_{i_s})$:

$$k = r_{i_0} + m_{i_0}\left(r_{i_1} + m_{i_1}\left(r_{i_2} + \cdots + m_{i_{s-1}}(r_{i_s} + m_{i_s}.0)\cdots\right)\right)$$

An iterative version

Input: $\mathcal{M} = \{m_1, \ldots, m_n\}$, $k \in \mathbb{N}$, $P \in E$

Output: $Q = [k]P \in E$

compute a randomized MRS representation of k

$Q \leftarrow \mathcal{O}$

for $j = s$ downto 0 **do**

$Q \leftarrow [m_{i_j}]Q + [r_{i_j}]P$

return Q

Game rules for designing robust algorithms

- **Simple attacks**
  Sequence of operations should be independent from the secret
- **Differential/Template attacks**
  Randomization strategies (scalar, point, coordinates, algorithm)
- **Timing attacks**
  Constant-time algorithms
- **Cache-timing attacks**
  No memory access at secret-data-dependent addresses
- **Fault attacks**
  No dummy operations

The Montgomery ladder

Input: $P, k = (k_{t-1} \dots k_0)_2$

Output: $[k]P$

$T_0 \leftarrow \mathcal{O}$

$T_1 \leftarrow P$

for $i = t - 1 \dots 0$ **do**

  if $k_i = 0$ **then**

    ec_add(T_1, T_0, T_1)

    ec_dbl(T_0, T_0)

  else $\quad \# \; k_i = 1$

    ec_add(T_0, T_0, T_1)

    ec_dbl(T_1, T_1)

return T_0

The Montgomery ladder

Input: $P, k = (k_{t-1} \dots k_0)_2$

Output: $[k]P$

$T_0 \leftarrow \mathcal{O}$

$T_1 \leftarrow P$

for $i = t - 1 \dots 0$ **do**

  if $k_i = 0$ **then**

    ec_add(T_1, T_0, T_1)

    ec_dbl(T_0, T_0)

  else $\quad \# \; k_i = 1$

    ec_add(T_0, T_0, T_1)

    ec_dbl(T_1, T_1)

return T_0

In each branch, operations are identical.
Only operands differ.

ec_add, **ec_dbl** may be combined into a single **ladderstep**

Invariant: $T_1 - T_0 = P$

Eliminating secret-data-dependant branches

Input: $P, k = (k_{t-1} \dots k_0)_2$

Output: $[k]P$

$T_0 \leftarrow \mathcal{O}$

$T_1 \leftarrow P$

$\sigma \leftarrow 0$

for $i = t - 1 \dots 0$ **do**

  if $k_i \oplus \sigma$ **then**

    swap(T_0, T_1)

  ec_add(T_1, T_0, T_1)

  ec_dbl(T_0, T_0)

  $\sigma \leftarrow k_i$

return $\sigma \,?\, T_1 : T_0$

Eliminating secret-data-dependant branches

Input: $P, k = (k_{t-1} \dots k_0)_2$

Output: $[k]P$

$T_0 \leftarrow \mathcal{O}$

$T_1 \leftarrow P$

$\sigma \leftarrow 0$

for $i = t - 1 \dots 0$ **do**

  if $k_i \oplus \sigma$ **then**

    swap(T_0, T_1)

  ec_add(T_1, T_0, T_1)

  ec_dbl(T_0, T_0)

  $\sigma \leftarrow k_i$

return $\sigma \,?\, T_1 : T_0$

Never use secret-data-dependent branch conditions

Replace:

if s **then**

  $r \leftarrow A$

else

  $r \leftarrow B$

by:

$r \leftarrow sA + (1 - s)B$

Leakage-free conditional instructions

Input: $P, k = (k_{t-1} \dots k_0)_2$

Output: $[k]P$

$T_0 \leftarrow \mathcal{O}$

$T_1 \leftarrow P$

$\sigma \leftarrow 0$

for $i = t - 1 \dots 0$ **do**

  cswap$(T_0, T_1, \sigma \oplus k_i)$

  ec_add(T_1, T_0, T_1)

  ec_dbl(T_0, T_0)

  $\sigma \leftarrow k_i$

cmove(T_0, T_1, σ)

return T_0

Use leakage-free conditional swap and move instructions

cmove(T_0, T_1, σ)
$T_0 \leftarrow T_0 \oplus ((T_0 \oplus T_1) \wedge \sigma)$

Trace: $(\mathtt{A}\,\mathtt{D})^+$

Double-scalar multiplication

Commonly used to compute
$[k_1]P + [k_2]Q$.

First proposed by Straus (1964).

Erroneously called Shamir's trick
since ElGamal's seminal paper
(1985).

Vector addition-subtraction chain

Double-scalar multiplication

Commonly used to compute $[k_1]P + [k_2]Q$.

First proposed by Straus (1964).

Erroneously called Shamir's trick since ElGamal's seminal paper (1985).

Vector addition-subtraction chain

Example: $[37]P + [22]Q$

$37 = 1\,0\,0\,1\,0\,1$

$22 = 0\,1\,0\,1\,1\,0$

$(1,1) = (1,0) + (0,1)$ precomp.

$(2,0) = 2.(1,0)$

$(2,1) = (2,0) + (0,1)$

$(4,2) = 2.(2,1)$

$(8,4) = 2.(4,2)$

$(9,5) = (8,4) + (1,1)$

$(18,10) = 2.(9,5)$

$(18,11) = (18,10) + (0,1)$

$(36,22) = 2.(18,11)$

$(37,22) = (36,22) + (1,0)$

Double-scalar multiplication

Commonly used to compute $[k_1]P + [k_2]Q$.

First proposed by Straus (1964).

Erroneously called Shamir's trick since ElGamal's seminal paper (1985).

Vector addition-subtraction chain

Not constant-time

Secret-data-dependent memory access

Example: $[37]P + [22]Q$

$37 = 1\,0\,0\,1\,0\,1$

$22 = 0\,1\,0\,1\,1\,0$

$(1,1) = (1,0) + (0,1)$ precomp.

$(2,0) = 2.(1,0)$

$(2,1) = (2,0) + (0,1)$

$(4,2) = 2.(2,1)$

$(8,4) = 2.(4,2)$

$(9,5) = (8,4) + (1,1)$

$(18,10) = 2.(9,5)$

$(18,11) = (18,10) + (0,1)$

$(36,22) = 2.(18,11)$

$(37,22) = (36,22) + (1,0)$

Computing $[m]Q + [r]P$ using a regular double ladder

Input: P, Q, m, r with $-m/2 \leq r < m/2$

Output: $[m]Q + [r]P$

$T_0 \leftarrow \mathcal{O}$

$T_1 \leftarrow Q$

ec_caddsub$(T_1, T_1, P, \text{sign}(r))$ # $T_1 \leftarrow Q \pm P$

$r \leftarrow r \times (1 - 2.\text{sign}(r))$ # $r \leftarrow |r|$

for $i = t - 1 \ldots 0$ **do**

ec_add(T_1, T_0, T_1)

ec_caddsub(T_0, T_1, P, r_i) # $T_0 \leftarrow T_1 \pm P$

cswap$(T_0, T_1, r_i \oplus m_i)$

ec_caddsub(T_0, T_0, Q, m_i) # $T_0 \leftarrow T_0 \pm Q$

$\sigma \leftarrow m_i$

cmove(T_0, T_1, σ)

return T_0

Invariant:
$T_1 - T_0 = Q \pm P$
depending on sign(r)

Computing $[k]P$ in constant-time

Input: $\mathcal{M} = \{m_1, \ldots, m_n\}$, $k \in \mathbb{N}, P \in E$
Output: $Q = [k]P \in E$

compute a randomized MRS representation of k
  # $k = r_{i_0} + m_{i_0}\left(r_{i_1} + m_{i_1}\left(r_{i_2} + \cdots + m_{i_{s-1}}(r_{i_s})\cdots\right)\right)$
$Q \leftarrow \mathcal{O}$
for $j = s$ downto 0 **do**
  $Q \leftarrow [m_{i_j}]Q + [r_{i_j}]P$ # Inner loop: regular double-ladder
return Q

Constant number of inner loops: choose moduli of the same size.

Computing $[k]P$ in constant-time

Input: $\mathcal{M} = \{m_1, \ldots, m_n\}$, $k \in \mathbb{N}, P \in E$
Output: $Q = [k]P \in E$

compute a randomized MRS representation of k
  # $k = r_{i_0} + m_{i_0}\left(r_{i_1} + m_{i_1}\left(r_{i_2} + \cdots + m_{i_{s-1}}(r_{i_s})\cdots\right)\right)$
$Q \leftarrow \mathcal{O}$
for $j = s$ downto 0 **do**
  $Q \leftarrow [m_{i_j}]Q + [r_{i_j}]P$ # Inner loop: regular double-ladder
return Q

Constant number of inner loops: choose moduli of the same size.

Constant number of outer loops: pick a random permutation of a fixed multiset $\{(m_1, e_1), \ldots, (m_n, e_n)\}$ and define $\alpha > 0$ such that:

$$\mathcal{P}' = \mathcal{P}/\min(m_i) \leq \alpha\ell < k + \alpha\ell < (\alpha+1)\ell \leq \prod_{j=1}^{n} m_j^{e_j} = \mathcal{P},$$

where ℓ is the order of P so that $[k]P = [k+\ell]P$ for all k.

Algorithm and execution trace

Input: $P, \mathcal{M} = \{m_1, \ldots, m_n\}, k, \ell$

Output: $[k]P$

Offline precomputations

Compute the multiset $\{(m_i, e_i)\}$ using a greedy approach and α such that $\mathcal{P}' \leq \alpha\ell < (\alpha + 1)\ell \leq \mathcal{P}$

number of MRS digits $w = \sum_i e_i$

Add $\alpha\ell$ to scalar k

Pick a random permutation of the moduli multiset

Compute the MRS representation of k, digit by digit

$k = ((m_0, r_0), \ldots, (m_{w-1}, r_{w-1}))$

Iteratively compute $Q \leftarrow [m_i]Q + [r_i]P$ using regular double ladder

Execution trace: * (A (A A A)... (A A A) D)^w

Level of randomization

The number of multiset permutations is given by:

$$\rho = \binom{w}{e_1, \ldots, e_n} = \frac{w!}{e_1! e_2! \ldots e_n!}, \qquad \text{where } w = \sum_{i=1}^{n} e_i.$$

Greedy approach for maximizing ρ:

Pick the m_i roundly until condition $\mathcal{P}' \leq \alpha\ell < (\alpha + 1)\ell \leq \mathcal{P}$ is fulfilled

Level of randomization

The number of multiset permutations is given by:

$$\rho = \binom{w}{e_1, \ldots, e_n} = \frac{w!}{e_1! e_2! \ldots e_n!}, \qquad \text{where } w = \sum_{i=1}^{n} e_i.$$

Greedy approach for maximizing ρ:

Pick the m_i roundly until condition $\mathcal{P}' \leq \alpha\ell < (\alpha+1)\ell \leq \mathcal{P}$ is fulfilled

Example: curve `Ed25519` [Bernstein'11]

$\ell = 2^{252} + 27742317777372353535851937790883648493$

$\mathcal{M} = \{30, 27, 25, 24, 20, 18\}$ (5-bit moduli)

$w = 56 \Rightarrow$ multiset is $\{(30,10),(27,10),(25,9),(24,9),(20,9),(18,9)\}$

$\alpha = 1 \Rightarrow k + \ell$ requires 56 MRS digits for all $0 < k < \ell$

$\rho = 311379875648763343856835700898158560000, \quad \log_2(\rho) \approx 131.19$

Experiments and comparisons

Constant-time randomized MRS algorithm

$\mathcal{M} = \{(30, 10), (27, 10), (25, 9), (24, 9), (20, 9), (18, 9)\}$

```
$ ./smul -sk 0 -sp 0 4 -t | awk -E ./cost.awk
k = 0x03b870822bd582698aed3bca2ff5607e4eb4daec66a10e5c59815a8142fb9fe0
P = (0x216936d3cd6e53fec0a4e231fdd6dc5c692cc7609525a7b2c9562d608f25d51a,
0x6666666666666666666666666666666666666666666666666666666666666658)
kP = (0x72e27e42122df1434121e3db8a0a5f8981dd5faed21b1c4b0bab64a46818aa74,
0x4839f85c0eb5a2161e6d50ed705c29f5ddff7fd6141cf631157829b516e2df60)
Cost:  6506M, 474S, 56m, 0d.
```

Experiments and comparisons

Constant-time randomized MRS algorithm

$\mathcal{M} = \{(30, 10), (27, 10), (25, 9), (24, 9), (20, 9), (18, 9)\}$

```
$ ./smul -sk 0 -sp 0 4 -t | awk -E ./cost.awk
k = 0x03b870822bd582698aed3bca2ff5607e4eb4daec66a10e5c59815a8142fb9fe0
P = (0x216936d3cd6e53fec0a4e231fdd6dc5c692cc7609525a7b2c9562d608f25d51a,
0x6666666666666666666666666666666666666666666666666666666666666658)
kP = (0x72e27e42122df1434121e3db8a0a5f8981dd5faed21b1c4b0bab64a46818aa74,
0x4839f85c0eb5a2161e6d50ed705c29f5ddff7fd6141cf631157829b516e2df60)
Cost:  6506M, 474S, 56m, 0d.
```

Constant-time Montgomery ladder (non-randomized)

```
$ ./smul -sk 0 -sp 0 1 -t | awk -E ./cost.awk
k = 0x03b870822bd582698aed3bca2ff5607e4eb4daec66a10e5c59815a8142fb9fe0
P = (0x216936d3cd6e53fec0a4e231fdd6dc5c692cc7609525a7b2c9562d608f25d51a,
0x6666666666666666666666666666666666666666666666666666666666666658)
kP = (0x72e27e42122df1434121e3db8a0a5f8981dd5faed21b1c4b0bab64a46818aa74,
0x4839f85c0eb5a2161e6d50ed705c29f5ddff7fd6141cf631157829b516e2df60)
Cost:  3303M, 1266S, 0m, 0d.
```

Experiments and comparisons (cont.)

Randomizing Montgomery ladder for curve `Ed25519`

Scalar randomizarion: compute $[k + r\ell]P$ for r random [Coron'99]

Curve `Ed25519` is defined modulo $p = 2^{255} - 19$ for fast arithmetic.

Hasse bound: the order of `Ed25519` is:

`0x1000000000000000000000000000000014def9dea2f79cd65812631a5cf5d3ed`

Thus r cannot be too small. Bernstein recommends a 256-bit integer.

Hence, a 512-bit Montgomery ladder scalar multiplication

`Cost: 6606M, 2532S, 0m, 0d.`

Constant-time randomized MRS algorithm

$\mathcal{M} = \{(31, 9), (30, 9), (29, 9), (28, 9), (27, 9), (26, 8)\}$

`Cost: 6158M, 462S, 53m, 0d.`

Approx. 25% saving

Game rules for designing robust algorithms

Simple attacks

sequence of operations should be independent from the secret ✔

Differential/Template attacks

Randomization strategies (scalar, point, coordinates, algorithm) ✔

Timing attacks

Algorithm must run in constant-time ✔

Cache-timing attacks

No memory access at secret-data-dependent addresses ✔

Fault attacks

No dummy operations ✔

Game rules for designing robust algorithms

Simple attacks

sequence of operations should be independent from the secret ✔

Differential/Template attacks

Randomization strategies (scalar, point, coordinates, algorithm) ✔

Timing attacks

Algorithm must run in constant-time ✔

Cache-timing attacks

No memory access at secret-data-dependent addresses ✔

Fault attacks

No dummy operations ✔

Check for faults

Horizontal attacks

Not so clear