# Breaking Randomized Mixed-Radix Scalar Multiplication Algorithms

Jérémie Detrey[1] Laurent Imbert[2]

[1]LORIA, Inria, CNRS, Univ. Lorraine, Nancy, France

[2]LIRMM, CNRS, Univ. Montpellier, France

Latincrypt 2019

Santiago de Chile – Oct. 2, 2019

# Context

Side-channel attacks and countermeasures for elliptic curve scalar multiplication: $k, P \mapsto [k]P = P + P + \cdots + P$

## Randomization strategies

- scalar blinding, point/cordinates randomization

# Context

Side-channel attacks and countermeasures for elliptic curve scalar multiplication: $k, P \mapsto [k]P = P + P + \cdots + P$

Randomization strategies

- scalar blinding, point/cordinates randomization

- randomized algorithms

  Idea: use a different, randomly selected addition chain for each scalar multiplication.

  - Ex: binary signed digits failures [Oswald, Aigner'01], [Ha, Moon'02].

# Context

Side-channel attacks and countermeasures for elliptic curve scalar multiplication: $k, P \mapsto [k]P = P + P + \cdots + P$

Randomization strategies

- scalar blinding, point/cordinates randomization

- randomized algorithms

  Idea: use a different, randomly selected addition chain for each scalar multiplication.

  - Ex: binary signed digits failures [Oswald, Aigner'01], [Ha, Moon'02].
  - Covering Systems of Congruences [Guerrini, I., Winterhalter'17]

# Today's talk

Covering systems of congruences

Full key recovery

A regular and constant-time generalization

A (virtual) template attack

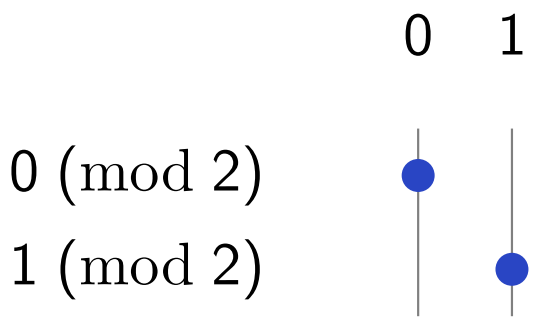
# Covering Systems of Congruences

A covering system of congruence (CSC) is a finite set of congruences $\mathcal{S} = \{r_i \bmod m_i\}_i$, s.t. every integer satisfies at least one of them.

# Covering Systems of Congruences

A covering system of congruence (CSC) is a finite set of congruences $\mathcal{S} = \{r_i \bmod m_i\}_i$, s.t. every integer satisfies at least one of them.

Example 1: binary decomposition




Binary aka double-and-add algorithm

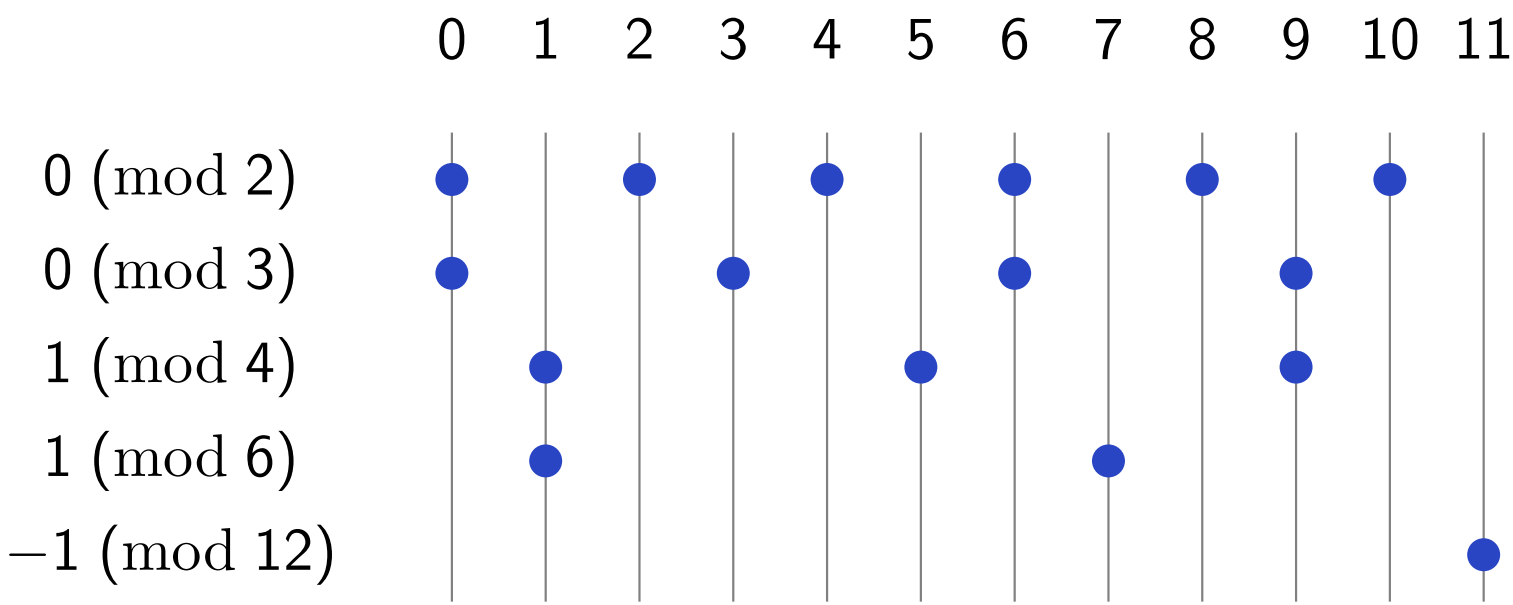
$k \equiv r \bmod 2 \Rightarrow [k]P = [2]Q + [r]P,$ where $Q = [(k - r)/2]P$

Not redundant $\Rightarrow$ non randomizable

# Covering Systems of Congruences

A covering system of congruence (CSC) is a finite set of congruences $\mathcal{S} = \{r_i \bmod m_i\}_i$, s.t. every integer satisfies at least one of them.

Example 2: multiple moduli




$k \equiv r \bmod m \Rightarrow [k]P = [m]Q + [r]P$, where $Q = [(k-r)/m]P$

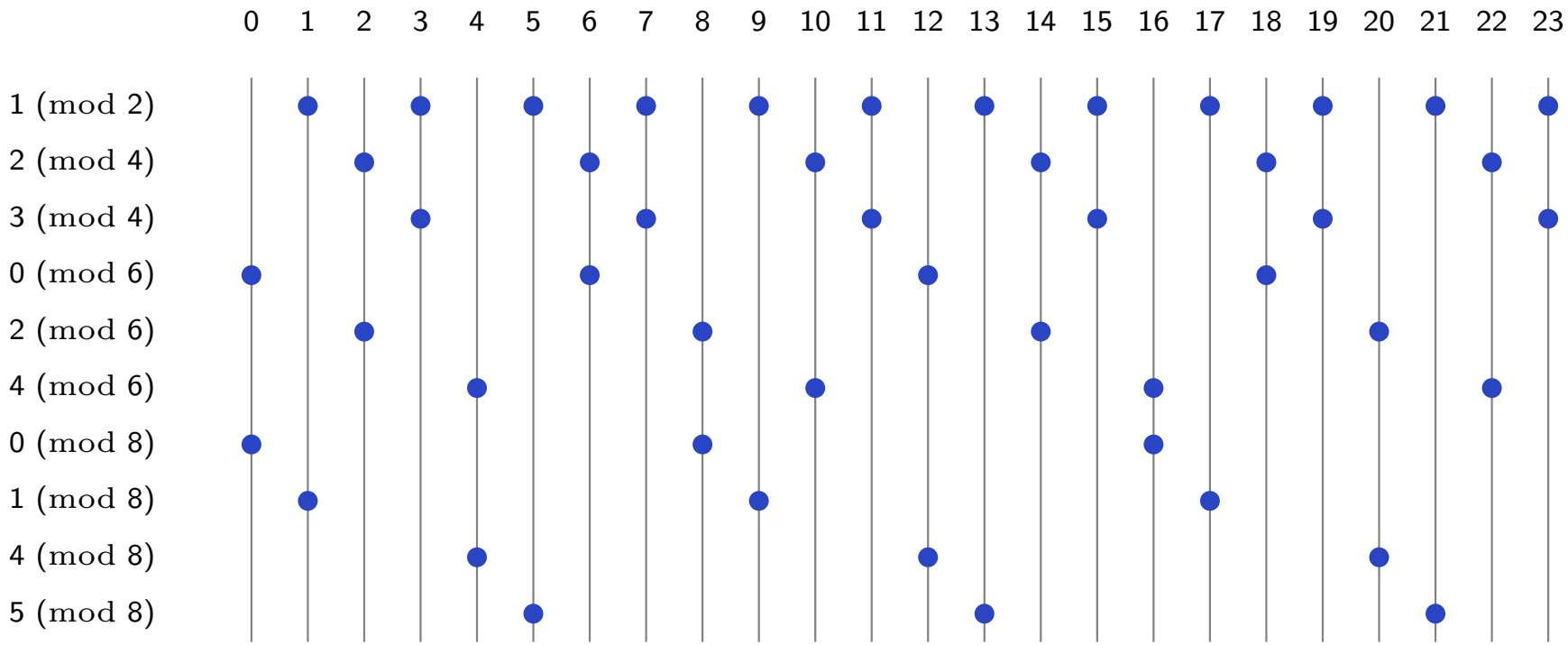
Redundant but not uniform

# Exact Coverings

A CSC is an *n*-cover if every integer is covered at least *n* times.

It is an exact *n*-cover if every integer is covered exactly *n* times.

Example: an exact 2-cover




Redundant and uniform

# A CSC-based Randomized Algorithm

**Input:** $\mathcal{S}$ an exact $n$-cover, $\ell = \mathrm{lcm}(m_1, \ldots, m_{|\mathcal{S}|})$, $k \in \mathbb{N}$, $P \in \mathcal{G}$

**Output:** $Q = [k]P \in \mathcal{G}$

1: **if** $k = 0$ **then**
2: **return** $\mathcal{O}$
3: **else if** $k = 1$ **then**
4: **return** $P$
5: Select $r \pmod m$ uniformly at random
among the $n$ classes that cover integers in $\ell\mathbb{Z} + k$
6: compute $Q \leftarrow [(k - r)/m]P$ recursively
7: **return** $[m]Q + [r]P$ # note: $[r]P$ may be precomputed

Covering systems of congruences

Full key recovery

A regular and constant-time generalization

A (virtual) template attack

# Threat model

The attacker can differentiate `D` from `A`.

Execution trace: concatenation of subtraces given by $[m]Q + [r]P$.

$$-1 \pmod 6 \longrightarrow [6]Q - P \longrightarrow \texttt{DADA}$$

$$2 \pmod{12} \longrightarrow [12]Q + [2]P \longrightarrow \texttt{DADDA} \quad ([2]P \text{ precomp.})$$

$$2 \pmod{12} \longrightarrow [2]([6]Q + P) \longrightarrow \texttt{DADAD}$$

# Threat model

The attacker can differentiate `D` from `A`.

Execution trace: concatenation of subtraces given by $[m]Q + [r]P$.

$$-1 \pmod 6 \longrightarrow [6]Q - P \longrightarrow \texttt{DADA}$$
$$2 \pmod{12} \longrightarrow [12]Q + [2]P \longrightarrow \texttt{DADDA} \quad ([2]P \text{ precomp.})$$
$$2 \pmod{12} \longrightarrow [2]([6]Q + P) \longrightarrow \texttt{DADAD}$$

$k =$ `0xfa72c39b25ecc4164d4c5ddeb506299c0941863eeee13f6d4d73fe32bfceec1f`

```
D D A D D D D D A D D D A D A D D A D A D D D A D D A D D A D D A D A D
D D A D D D A D D D D D D A D D D D D A D A D A D A D A D D A D D A D D
A D D D D D A D D D A D D A D A D A D D D D D A D A D A D D A D A D D D
A D A D D A D D D A D A D D D D A D D A D D D A D D A D A D A D D D D A
D D D A D D A D D D D D D D D A D A D D D D A D A D A D D A D A D D A D
D D D D A D D D D A D A D D D A D A D A D A D D D A D A D A D D D D A D
D D A D D D D A D A D A D D A D A D A D D D D D D D A D D D D D D A D A
D A D D A D D D A D A D D A D D D A D A D A D A D D D D A D D A D D A D
A D D D D A D D D A D A D A D D D D D D D A D D A D D D A D A D D A D A
D A D A D D A D D A D A D D D D A D A D D D A D A D A D D A D A D D A D
```

Randomization provides a huge number of traces for a given $k$.

# (Weak) security assumption

The mapping $Tr$ from $\mathbb{Z}$ to $(\mathtt{D}|\mathtt{A})*$ is not injective.

$$10273 = 1 + 12(0 + 4(10 + 12(5 + 12(1 + 12.0))))$$
$$Tr(10273) = \mathtt{D\,A\,D\,D\,A\,D\,A\,D\,D\,A\,D\,A\,D\,D\,A\,D\,D\,D\,A\,D\,D\,A}$$

$$43455 = 3 + 4(7 + 8(1 + 12(5 + 12(9 + 12.0)))),$$
$$Tr(43455) = \mathtt{D\,A\,D\,D\,A\,D\,A\,D\,D\,A\,D\,A\,D\,D\,A\,D\,D\,D\,A\,D\,D\,A}$$

$$14649 = 9 + 12(0 + 4(5 + 12(1 + 12(2 + 12.0)))),$$
$$Tr(14649) = \mathtt{D\,A\,D\,D\,A\,D\,A\,D\,D\,A\,D\,A\,D\,D\,A\,D\,D\,D\,A\,D\,D\,A}$$

Empirical estimate: #integers that maps to a given trace(*) $> 2^{116}$

(*) of length equal to the average length of a trace produced by 256-bit integers

# The mapping $Tr^{-1}$

Example for u3c-48-24

| | |
|---|---|
| D | { ( 0, 2) } |
| DD | { ( 0, 4) } |
| DDA | { (-1, 4) } |
| DDD | { ( 0, 8) } |
| DADA | { ( 3, 6), (-1, 6), ( 1, 6) } |
| DDAD | { (-2, 8) } |
| DDDA | { (-1, 8), ( 1, 8) } |
| DADAD | { (-2, 12), ( 2, 12), ( 6, 12) } |
| DADDA | { ( 1, 12), ( 5, 12) } |
| DDADA | { (-3, 12) } |
| DDADD | { ( 4, 16), (-4, 16) } |
| DDDAD | { ( 2, 16), (-6, 16) } |
| DDDDA | { ( 5, 16), (-3, 16), (-5, 16), (3, 16) } |

# Full key recovery algorithm (on a toy example)

T1: `DDADD D DDA D DD` (split up for simplification)

# Full key recovery algorithm (on a toy example)

T1: `DDADD D DDA D DD` (split up for simplification)

`DDADD D DDA D --` $0 \pmod 4 \quad \Rightarrow \quad k \in 4\mathbb{Z}$

# Full key recovery algorithm (on a toy example)

T1: `DDADD D DDA D DD` (split up for simplification)

| | | | |
|---|---|---|---|
| `DDADD D DDA D --` | $0 \pmod 4$ | $\Rightarrow$ | $k \in 4\mathbb{Z}$ |
| `DDADD D DDA - --` | $0 \pmod 2$ | $\Rightarrow$ | $k \in 8\mathbb{Z}$ |

# Full key recovery algorithm (on a toy example)

T1: `DDADD D DDA D DD` (split up for simplification)

| | | | |
|---|---|---|---|
| `DDADD D DDA D --` | $0 \pmod 4$ | $\Rightarrow$ | $k \in 4\mathbb{Z}$ |
| `DDADD D DDA - --` | $0 \pmod 2$ | $\Rightarrow$ | $k \in 8\mathbb{Z}$ |
| `DDADD D --- - --` | $-1 \pmod 4$ | $\Rightarrow$ | $k \in 32\mathbb{Z} - 8$ |

# Full key recovery algorithm (on a toy example)

T1: `DDADD D DDA D DD` (split up for simplification)

| | | | |
|---|---|---|---|
| `DDADD D DDA D --` | $0 \pmod 4$ | $\Rightarrow$ | $k \in 4\mathbb{Z}$ |
| `DDADD D DDA - --` | $0 \pmod 2$ | $\Rightarrow$ | $k \in 8\mathbb{Z}$ |
| `DDADD D --- - --` | $-1 \pmod 4$ | $\Rightarrow$ | $k \in 32\mathbb{Z} - 8$ |
| `DDADD - --- - --` | $0 \pmod 2$ | $\Rightarrow$ | $k \in 64\mathbb{Z} - 8$ |

# Full key recovery algorithm (on a toy example)

T1: `DDADD D DDA D DD` (split up for simplification)

| | | | |
|---|---|---|---|
| `DDADD D DDA D --` | $0 \pmod 4$ | $\Rightarrow$ | $k \in 4\mathbb{Z}$ |
| `DDADD D DDA - --` | $0 \pmod 2$ | $\Rightarrow$ | $k \in 8\mathbb{Z}$ |
| `DDADD D --- - --` | $-1 \pmod 4$ | $\Rightarrow$ | $k \in 32\mathbb{Z} - 8$ |
| `DDADD - --- - --` | $0 \pmod 2$ | $\Rightarrow$ | $k \in 64\mathbb{Z} - 8$ |
| `----- - --- - --` | $4 \pmod{16}$ | $\Rightarrow$ | $k \in 1024\mathbb{Z} + 248$ |
| | $-4 \pmod{16}$ | $\Rightarrow$ | $k \in 1024\mathbb{Z} - 264$ |

# Full key recovery algorithm (on a toy example)

T1: `DDADD D DDA D DD`

`----- - --- - --`

$$\begin{aligned} 4 \pmod{16} &\Rightarrow k \in 1024\mathbb{Z} + 248 \\ -4 \pmod{16} &\Rightarrow k \in 1024\mathbb{Z} - 264 \end{aligned}$$

T2: `DDDAD D D DDAD DD`

| | | | |
|---|---|---|---|
| `DDDAD D D DDAD --` | $0 \pmod 4$ | $\Rightarrow$ | $k \in 4\mathbb{Z}$ |
| `DDDAD D D ---- --` | $-2 \pmod 8$ | $\Rightarrow$ | $k \in 32\mathbb{Z} - 8$ |
| `DDDAD D - ---- --` | $0 \pmod 2$ | $\Rightarrow$ | $k \in 64\mathbb{Z} - 8$ |
| `DDDAD - - ---- --` | $0 \pmod 2$ | $\Rightarrow$ | $k \in 128\mathbb{Z} - 8$ |
| `----- - - ---- --` | $2 \pmod{16}$ | $\Rightarrow$ | $k \in 2048\mathbb{Z} + 248$ |
| | $-6 \pmod{16}$ | $\Rightarrow$ | $k \in 2048\mathbb{Z} - 776$ |

# Full key recovery algorithm (on a toy example)

T1: `DDADD D DDA D DD`

`_____ _ ___ _ __`

$$\begin{aligned} 4 \pmod{16} &\Rightarrow k \in 1024\mathbb{Z} + 248 \\ -4 \pmod{16} &\Rightarrow k \in 1024\mathbb{Z} - 264 \end{aligned}$$

T2: `DDDAD D D DDAD DD`

`_____ _ _ ____ __`

$$\begin{aligned} 2 \pmod{16} &\Rightarrow k \in 2048\mathbb{Z} + 248 \\ -6 \pmod{16} &\Rightarrow k \in 2048\mathbb{Z} - 776 \end{aligned}$$

# Full key recovery algorithm (on a toy example)

T1: `DDADD D DDA D DD`

`----- - --- - --`

$$4 \pmod{16} \quad \Rightarrow \quad k \in 1024\mathbb{Z} + 248$$
$$\cancel{-4 \pmod{16} \quad \Rightarrow \quad k \in 1024\mathbb{Z} - 264}$$

T2: `DDDAD D D DDAD DD`

`----- - - ---- --`

$$2 \pmod{16} \quad \Rightarrow \quad k \in 2048\mathbb{Z} + 248$$
$$-6 \pmod{16} \quad \Rightarrow \quad k \in 2048\mathbb{Z} - 776$$

# Full key recovery algorithm (on a toy example)

T1: `DDADD D DDA D DD`

`_____ _ ___ _ __`

$$\begin{aligned} 4 \pmod{16} &\Rightarrow k \in 1024\mathbb{Z} + 248 \\ \cancel{-4 \pmod{16}} &\Rightarrow \cancel{k \in 1024\mathbb{Z} - 264} \end{aligned}$$

T2: `DDDAD D D DDAD DD`

`_____ _ _ ____ __`

$$\begin{aligned} 2 \pmod{16} &\Rightarrow k \in 2048\mathbb{Z} + 248 \\ -6 \pmod{16} &\Rightarrow k \in 2048\mathbb{Z} - 776 \end{aligned}$$

- Pruning strategy to limit exponential growth of partially decoded traces
- Work without preliminary splitting
- Work when $[r]P$ is precomputed (traces only reveal $m$-values)
- Can recover the whole scalar with very few traces

Covering systems of congruences

Full key recovery

A regular and constant-time generalization

A (virtual) template attack

# Mixed-radix number system

Write $k$ in base $(b_1, \ldots, b_n)$ s.t.: ($b_i$ need not be distincts)

$$k = k_1 + b_1 \left(k_2 + b_2 \left(k_3 + \cdots + b_n (k_{n+1}) \cdots \right)\right)$$

# Mixed-radix number system

Write $k$ in base $(b_1, \ldots, b_n)$ s.t.: ($b_i$ need not be distincts)

$$k = k_1 + b_1 \left(k_2 + b_2 \left(k_3 + \cdots + b_n (k_{n+1}) \cdots \right)\right)$$

## Randomized MRS-based scalar multiplication

**Input:** A family $\mathcal{F}$ of "good" MRS bases, $k \in \mathbb{Z}$ and $P \in E$.

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

$\mathcal{B} = (b_1, \ldots, b_n) \xleftarrow{\$} \mathcal{F}$

compute the MRS representation of $k$ in $\mathcal{B}$

$Q \leftarrow [k_{n+1}]P$ # Montgomery ladder

**for** $j = n$ downto 1 **do**

$Q \leftarrow [b_i]Q + [k_i]P$

**return** $Q$

Generalizes covering systems of congruences

# Loading the bases

Requirements for suitable families of MRS bases:

1. the range $\{X_{\mathcal{F}}^{\min}, \ldots, X_{\mathcal{F}}^{\max}\}$ can accommodate any scalar $k$ of relevance for the cryptosystem at hand
2. $\mathcal{F}$ is large enough to ensure a sufficient amount of randomization
3. one can securely compute the MRS representation of $k$ in base $\mathcal{B}$
4. one can securely compute the scalar multiplication $[k]P$

# The family $\mathcal{F}_{s,n,m}$

The set of all $n$-tuples exclusively composed of $s$-bit moduli taken from a predefined set of size $m$

Provides $\rho = \lfloor \log_2(m^n) \rfloor$ bits of randomization

Among all possible sets of size $m$, those composed of the $m$ largest $s$-bit integers provide shorter MRS representations.

$$\mathcal{F}_{s,n,m} = \{(b_1, \ldots, b_n) \ : \ 2^s - m \leq b_i \leq 2^s - 1 \text{ for } 1 \leq i \leq n\}.$$

Example:

$s = 5$ bits, $m = 7$

$\hookrightarrow b_i \in \{25, 26, 27, 28, 29, 30, 31\}$

$n = 53$

$\hookrightarrow m^n > 2^{128}$

# Computing $[k]P$ securely

- Compute $[b_i]Q + [k_i]P$ in constant time for all $(b_i, k_i)$
- Compute the MRS representation of $k$ in constant time
- Ensure a constant number of iterations
- No secret-dependent branching nor memory access

# Computing $[k]P$ securely

- Compute $[b_i]Q + [k_i]P$ in constant time for all $(b_i, k_i)$
  - All moduli $b_i$ have a fixed size $s$
  - Regular double ladder similar to [Bernstein'06]
    $\hookrightarrow$ trace: $\underline{\mathsf{A}\,\mathsf{D}\,\mathsf{A}}\ \underline{\mathsf{A}\,\mathsf{D}\,\mathsf{A}}\ \ldots\ \underline{\mathsf{A}\,\mathsf{D}\,\mathsf{A}}$ ($s$ times)

- Compute the MRS representation of $k$ in constant time

- Ensure a constant number of iterations

- No secret-dependent branching nor memory access

# Computing $[k]P$ securely

- Compute $[b_i]Q + [k_i]P$ in constant time for all $(b_i, k_i)$
  - All moduli $b_i$ have a fixed size $s$
  - Regular double ladder similar to [Bernstein'06]
    $\hookrightarrow$ trace: $\underline{\mathsf{A}\,\mathsf{D}\,\mathsf{A}}\ \underline{\mathsf{A}\,\mathsf{D}\,\mathsf{A}}\ \ldots\ \underline{\mathsf{A}\,\mathsf{D}\,\mathsf{A}}$ ($s$ times)

- Compute the MRS representation of $k$ in constant time
  - Euclidean division by constants through multiplication by precomputed inverses [Grandlund, Montgomery'94]

- Ensure a constant number of iterations

- No secret-dependent branching nor memory access

# Computing $[k]P$ securely

- Compute $[b_i]Q + [k_i]P$ in constant time for all $(b_i, k_i)$
  - All moduli $b_i$ have a fixed size $s$
  - Regular double ladder similar to [Bernstein'06]
    $\hookrightarrow$ trace: $\underline{\mathsf{A\,D\,A}}\ \underline{\mathsf{A\,D\,A}}\ \ldots\ \underline{\mathsf{A\,D\,A}}$ ($s$ times)

- Compute the MRS representation of $k$ in constant time
  - Euclidean division by constants through multiplication by precomputed inverses [Grandlund, Montgomery'94]

- Ensure a constant number of iterations
  - $\hat{k} = k + \alpha \times \#E$ for a computable value $\alpha$ (independent of $k$) s.t. $\hat{k}$ can be written in any base $\mathcal{B} \in \mathcal{F}$ using exactly $n + 1$ digits

- No secret-dependent branching nor memory access

Covering systems of congruences

Full key recovery

A regular and constant-time generalization

A (virtual) template attack

# Creating the template

Public information: $P \in E$ of order $\#E$.

Hardware model: the adversary has full access to a training device. She knows the public key $R = [k]P$.

# Creating the template

Public information: $P \in E$ of order $\#E$.

Hardware model: the adversary has full access to a training device. She knows the public key $R = [k]P$.

- For all $b \in [2^s - m, 2^s - 1]$ and all $i = 0, \ldots, b-1$, precompute points $R_{b,i} = [b^{-1} \bmod \#E](R - [i]P)$.

  Invariant: $[b]R_{b,i} + [i]P = R$

# Creating the template

Public information: $P \in E$ of order $\#E$.

Hardware model: the adversary has full access to a training device. She knows the public key $R = [k]P$.

- For all $b \in [2^s - m, 2^s - 1]$ and all $i = 0, \ldots, b-1$, precompute points $R_{b,i} = [b^{-1} \bmod \#E](R - [i]P)$.

  Invariant: $[b]R_{b,i} + [i]P = R$

- Reprogram the training device to evaluates $[b]R_{b,i} + [i]P$ for all $b$ and $i$ and store the $m \times 2^s$ corresponding traces.

  Sequences of operations are identical but bit-flips differ.

  Deep-learning or other advanced techniques may allow to differentiate these traces.

# The attack

- Observe that the algorithm always ends after computing $[b]R_{b,i} + [i]P = [k]P$, for one of the precomputed point $R_{b,i}$ and $i = k \bmod b$.
- On the attacked device, record traces for $[k]P$.
  Running the algorithm sufficiently many times provides $m$ different traces, one for for each $b$-value.
- Apply the template to recover $k \bmod b$ for all $b \in [2^s - m, 2^s - 1]$.
- Finally, use the CRT to get $k \bmod \mathrm{lcm}(2^s - m, \ldots, 2^s - 1)$.

Example: $s = 8$, $m = 16$, $3960 < 2^{12}$ traces, $\mathrm{lcm}(240, \ldots, 255) > 2^{92}$.

# Conclusions

- Covering systems randomization: broken
- Constant-time mixed-radix randomization

# Conclusions

- Covering systems randomization: broken
- Constant-time mixed-radix randomization
- May succomb a virtual template attack

  It exploits an inherent weakness of MRS representation. Namely, the fact that $k_1 (= k \bmod b_1)$ solely depends on $b_1$.

  Although $b_1$ is an $s$-bit integer, randomization implies that $b_1$ takes many different values, allowing the attacker to recover much more than $s$ bits of $k$ thanks to the CRT.

# Conclusions

- Covering systems randomization: broken
- Constant-time mixed-radix randomization
- May succomb a virtual template attack

  It exploits an inherent weakness of MRS representation. Namely, the fact that $k_1 (= k \bmod b_1)$ solely depends on $b_1$.

  Although $b_1$ is an $s$-bit integer, randomization implies that $b_1$ takes many different values, allowing the attacker to recover much more than $s$ bits of $k$ thanks to the CRT.
- Can we effectively create the template?

  Can we recover the whole key using Coppersmith-like techniques?

# Conclusions

- Covering systems randomization: broken
- Constant-time mixed-radix randomization
- May succomb a virtual template attack

  It exploits an inherent weakness of MRS representation. Namely, the fact that $k_1 (= k \bmod b_1)$ solely depends on $b_1$.

  Although $b_1$ is an $s$-bit integer, randomization implies that $b_1$ takes many different values, allowing the attacker to recover much more than $s$ bits of $k$ thanks to the CRT.
- Can we effectively create the template?

  Can we recover the whole key using Coppersmith-like techniques?
- Yet, we do not recommend the use of MRS as a randomization setting.