SDU
University of
Southern Denmark

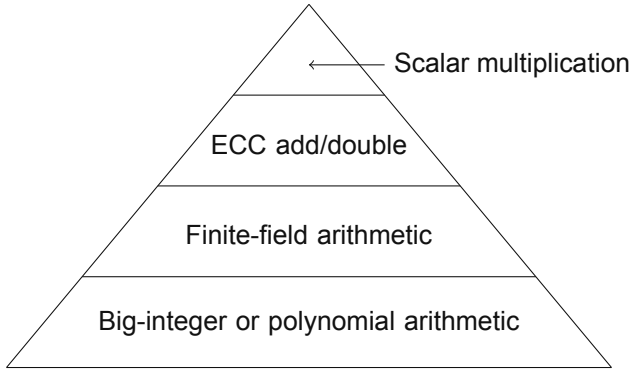
Cryptographic Engineering

Optimizing Elliptic-Curve Arithmetic

Ruben Niederhagen
(based on content by Peter Schwabe and Lejla Batina)

Department of Mathematics and Computer Science (IMADA)

The ECC Pyramid



The top of the pyramid:

- Pyramid levels are not independent.
- Interactions trough all levels, relevant for:
  - correctness,
  - security, and
  - performance.
- Setting for this lecture (peak of the pyramid):
  - Consider (finite, abelian) group $\mathcal{G}$, written additively.
  - Compute $k \cdot P$ for $k \in \mathbb{Z}$ and $P \in \mathcal{G}$.
  - This is the same as x^k for x in a multiplicative group $\mathcal{G}$.
  - Same algorithms for scalar multiplication and exponentiation.

Definition

Given two points P and Q on an elliptic curve, such that $Q \in \langle P \rangle$, find an integer k such that $kP = Q$.

- Typical setting for cryptosystems:
  - P is a fixed system parameter,
  - k is the secret (private) key,
  - Q is the public key.
- Key generation needs to compute $Q = kP$, given k and P.

Department of Mathematics
and Computer Science (IMADA)

- Users Alice and Bob agree on a curve and a base point P on the curve.
- Alice and Bob generate key pairs $(k_A, Q_A = k_A P)$ and $(k_B, Q_B = k_B P)$.
- Alice sends Q_A to Bob.
- Bob sends Q_B to Alice.
- Alice computes joint key as $K = k_A Q_B = k_A k_B P = (k_A k_B)P$.
- Bob computes joint key as $K = k_B Q_A = k_B k_A P = (k_A k_B)P$.

- Alice has key pair $(k_A, Q_A = k_A P)$.
- Order of $\langle P \rangle$ is ℓ; use cryptographic hash function H().

Sign: Generate secret random $r \in \{1, \ldots, \ell\}$,
compute signature (h, s) on message M as:

$$h = \mathrm{H}(rP|M)$$

$$s = (r - hk_A) \mod \ell$$

Verify: Compute:

$$\begin{aligned} R &= sP + hQ_A = sP + h(k_A P) \\ &= (s + hk_A)P = (r - hk_A + hk_A)P \\ &= rP \end{aligned}$$

and check that $\mathrm{H}(R|M) = h$.

Department of Mathematics
and Computer Science (IMADA)

- Looks like all these schemes need computation of kP.
- Let's take a closer look:
  - For key generation, the point P is *fixed* at compile time.
  - For Diffie-Hellman joint-key computation the point is received at runtime.
  - Key generation and Diffie-Hellman need one scalar multiplication kP.
  - Schnorr signature verification needs double-scalar multiplication $k_1P_1 + k_2P_2$.
  - In key generation and Diffie-Hellman joint-key computation, k is secret.
  - The scalars in Schnorr signature verification are public.
- In the following: Distinguish these cases.

Department of Mathematics
and Computer Science (IMADA)

- Let's compute $105 \cdot P$.
- Obvious: Can do that with 104 additions $P + P + P + \cdots + P$.
- Problem: $105 = 1101001_b$ has 7 bits, we need roughly 2^7 additions, *cryptographic* scalars have ≈ 256 bits, we would need roughly 2^{256} additions. (More expensive than solving the ECDLP!)
- Conclusion: we need algorithms that run in polynomial time (in the size of the scalar).

Department of Mathematics
and Computer Science (IMADA)

$$
\begin{aligned}
105 &= 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0 \\
&= 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\
&= ((((((((((1 \cdot 2 + 1) \cdot 2) + 0) \cdot 2) + 1) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + 1 \quad \text{(Horner's rule)}
\end{aligned}
$$

$$
105 \cdot P = ((((((((((P \cdot 2 + P) \cdot 2) + 0) \cdot 2) + P) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + P
$$

$\Rightarrow$ Cost: 6 doublings, 3 additions.

General Algorithm: ``Double and Add''

Require: Point P, scalar factor k.
Ensure: $R = kP$.
  $R \leftarrow P$
  for $i \leftarrow n-2$ **downto** 0 **do**
    $R \leftarrow 2R$ /* double */
    if $(k)_2[i] = 1$ **then**
      $R \leftarrow R + P$ /* add */
    end if
  end for
  return R

Analysis of Double-and-Add

- Let n be the number of bits in the exponent.
- Double-and-add takes $n - 1$ doublings.
- Let m be the number of 1 bits in the exponent.
- Double-and-add takes $m - 1$ additions.
- On average: $m \approx n/2$ additions.
- P does not need to be known in advance, no precomputation depending on P.
- Handles single-scalar multiplication.
- Running time clearly depends on the scalar.

$\Rightarrow$ Insecure for secret scalars!

Double-Scalar Double-and-Add

- Let's modify the algorithm to compute $k_1P_1 + k_2P_2$.
- Obvious solution:
  - Compute k_1P_1 ($n_1 - 1$ dbl, $m_1 - 1$ add).
  - Compute k_2P_2 ($n_2 - 1$ dbl, $m_2 - 1$ add).
  - Add the results (1 addition).
- We can do better: $\longrightarrow$
- $\max(n_1, n_2)$ dbl, $m_1 + m_2$ add.

Require: Points P_1 and P_2,
  scalar factors k_1 and k_2.
Ensure: $R = k_1P_1 + k_2P_2$.
  $R \leftarrow \mathcal{O}$
  for $i \leftarrow \max(n_1, n_2) - 1$ **downto** 0 **do**
    $R \leftarrow 2R$ /* double */
    if $(k_1)_2[i] = 1$ **then**
      $R \leftarrow R + P_1$ /* add */
    end if
    if $(k_2)_2[i] = 1$ **then**
      $R \leftarrow R + P_2$ /* add */
    end if
  end for
  return R

Some precomputation helps:

- Whenever k_1 and k_2 have a 1 bit at the same position, we first add P_1 and then P_2 (on average for $1/4$ of the bits).
- Let's just precompute $T = P_1 + P_2$.
- Modified algorithm (special case of Strauss' algorithm): $\longrightarrow$

$R \leftarrow \mathcal{O}$; $T \leftarrow P_1 + P_2$
for $i \leftarrow \max(n_1, n_2) - 1$ **downto** 0 **do**
  $R \leftarrow 2R$ /* double */
  if $(k_1)_2[i] = 1$ *AND* $(k_2)_2[i] = 1$ **then**
    $R \leftarrow R + T$ /* add */
  else
    if $(k_1)_2[i] = 1$ **then**
      $R \leftarrow R + P_1$ /* add */
    end if
    if $(k_2)_2[i] = 1$ **then**
      $R \leftarrow R + P_2$ /* add */
    end if
  end if
end for
return R

Even more (offline) precomputation:

- What if precomputation is free (fixed basepoint, offline precomputation)?
- First idea: Let's precompute a table containing $(0P, 1P, 2P, 3P, \dots)$; when we receive k, simply look up kP.
- Problem: k is large. For a 256-bit k this is totally impractical.
- How about, for example, precompute $(P, 2P, 4P, 8P, \dots, 2^{n-1}P)$.
- This needs only about 16kB of storage for $n = 256$ and 64-byte group elements.

- Modified scalar-multiplication algorithm:

Require: Point P, scalar factor k.
Ensure: $R = kP$.
  $R \leftarrow \mathcal{O}$
  for $i \leftarrow 0$ **to** $n - 1$ **do**
    if $(k)_2[i]$ **then**
      $R \leftarrow R + 2^i P$
    end if
  end for
  return R

Eliminated all doublings in fixed-basepoint scalar multiplication!

Double-and-Add Always

- All algorithms so far perform *conditional addition* where the condition is secret.
- For secret scalars (most common case!) we need something else.
- Idea: Always perform addition, discard result.
- Or simply add the neutral element $\mathcal{O}$.
- **Still not constant time, more later**...

Require: Point P, scalar factor k.
Ensure: $R = kP$.

```
R ← P
for i ← n − 2 downto 0 do
    R ← 2R
    if (k)_2[i] = 1 then
        R ← R + P
    else
        R ← R + O
    end if
end for
return R
```

Let's rewrite that a bit:

- We have a table $T = (\mathcal{O}, P)$.
- Notation $T[0] = \mathcal{O}$, $T[1] = P$.
- Scalar multiplication is:

  Require: Point P, scalar factor k.
  Ensure: $R = kP$.
  $R \leftarrow P$
  for $i \leftarrow n - 2$ **downto** 0 **do**
  $R \leftarrow 2R$
  $R \leftarrow R + T[(k)_2[i]]$
  end for

Changing the Scalar Radix

- So far we considered a scalar written in radix 2.
- How about radix 3?
- We precompute a Table $T = (\mathcal{O}, P, 2P)$
- Write scalar k as $(k_{n-1}, \ldots, k_0)_3$.
- Compute scalar multiplication as:

  $R \leftarrow T[(k)_3[n-1]]$
  for $i \leftarrow \log_3(k) - 2$ **downto** 0 **do**
    $R \leftarrow 3R$
    $R \leftarrow R + T[(k)_3[i]]$
  end for
- Advantage: The scalar is shorter, fewer additions.
- Disadvantage: 3 is just not nice (needs triplings).
- How about some nice numbers, like 4, 8, 16?

Fixed-Window Scalar Multiplication

- Fix a window width w.
- Precompute $T = (\mathcal{O}, P, 2P, \ldots, (2^w - 1)P)$.
- Write scalar k as $(k_{m-1}, \ldots, k_0)_{2^w}$.
- This is the same as chopping the binary scalar into "windows" of fixed width w.

Require: Point P, scalar factor k,
  $m = \log_{2^w}(k)$.
Ensure: $R = kP$.
  $R \leftarrow T[(k)_{2^w}[m-1]]$
  for $i \leftarrow m-2$ **downto** 0 **do**
    for $j \leftarrow 1$ **to** w **do**
      $R \leftarrow 2R$
    end for
    $R \leftarrow R + T[(k)_{2^w}[i]]$
  end for
  return R

Analysis of Fixed-Window Scalar Mult

- For an n-bit scalar we still have $n - 1$ doublings.
- Precomputation costs $2^w/2 - 1$ additions and $2^w/2 - 1$ doublings.
- Number of additions in the loop is $\lceil n/w \rceil$.
- Larger w: More precomputation.
- Smaller w: More additions inside the loop.
- For ≈ 256-bit scalars choose $w = 4$ or $w = 5$.

Is fixed-window *constant time*?

- For each window of the scalar perform w doublings and one addition, sounds good.
- The devil is in the detail:
  - Is addition running in constant time? Also for $\mathcal{O}$?
  - We can make that work, but how easy and efficient it is depends on the curve shape (remember tricky cases for fast addition on Weierstrass curves).
  - Remember that table lookups are generally not constant time!

Making it constant time:


```
/* For point P contains pre-computed multiples 0, P, 2*P, 3*P,...,255*P */
extern ec_point precomputed[255];

void ec_scalarmult_P(ec_point *r, unsigned char scalar[32]) {
  int i,j;
  ec_point t;
  ec_setneutral(r);
  for (i = 31; i >= 0; i--) {
    for (j=0; j<8; j++)
      ec_point_double(r, r);
    ec_point_lookup(&t, precomputed, scalar[i]);
    ec_point_add(r, r, &t);
  }
}
```

```
static void ec_point_lookup(ec_point *t, const ec_point *table, int pos)
{
  int i,j;
  unsigned char b;
  *t = table[0];
  for (i = 0; i < 256; i++)
  {
    b = int_eq(i, pos);             // set b=1 if i==pos, else set b=0
    ec_point_cmov(t, table[i], b);  // Copy table[i] to t if b is 1
  }
}
```

`int_eq` and `ec_point_cmov`


```
unsigned char int_eq(int a, int b)
{
  unsigned long long t = a ^ b;
  t = (-t) >> 63;
  return 1-t;
}
```

```
void ec_point_cmov(ec_point *r,
        const ec_point *t, unsigned char b)
{
  unsigned char *u = (unsigned char *)r;
  unsigned char *v = (unsigned char *)t;
  int i;
  b = -b;
  for (i = 0; i < sizeof(ec_point); i++)
    u[i] = (b & v[i]) ^ (~b & u[i]);
}
```

More Offline Precomputation

- Let's get back to fixed-basepoint multiplication.
- So far we precomputed $P, 2P, 4P, 8P, \ldots$.
- We can combine that with fixed-window scalar multiplication.
- Precompute $T_i = (\mathcal{O}, P, 2P, 3P, \ldots, (2^w - 1)P) \cdot 2^i$ for $i = 0, w, 2w, 3w, \lceil n/w \rceil - 1$.
- Perform scalar multiplication as:

  $R \leftarrow T_0[(k)_{2^w}[0]]$
  for $i \leftarrow 1$ **to** $\lceil n/w \rceil - 1$ **do**
    $R \leftarrow R + T_{iw}[(k)_{2^w}[i]]$
  end for

- No doublings, only $\lceil n/w \rceil - 1$ additions.
- Can use huge w, but:
  - at some point the precomputed tables don't fit into cache anymore, and
  - constant-time loads get slow for large w.

Fixed-Window Limitations

Consider the scalar $22 = (1\,01\,10)_2$ **and window size 2:**

$1\,01\,10$

- Initialize R with P.
- Double, double, add P.
- Double, double, add $2P$.

More efficient:

$10\,11\,0$

- Initialize R with P.
- Double, double, double, add $3P$.
- Double.

- Problem with fixed window: It is fixed.
- Idea: “Slide” the window over the scalar.

Sliding-Window Scalar Multiplication

- Choose window size w.
- Rewrite scalar k' as $k = (k_m, \ldots, k_0)$ with k_i in $W = \{2^{w-1}, 2^{w-1}+1, \ldots, 2^w - 1\}$, e.g., for $w = 4$:

$$k' = (1010011010011000)_b$$

$$k = (1010\ 0\ 1101\ 0\ 0\ 1100\ 0)_b$$

- Do this by scanning k from right to left, expand window from each 1-bit.
- Precompute wP for all $w \in W$.

Require: Scalar factor $k = (k_m, \ldots, k_0)$.
Ensure: $R = kP$.

```
R ← k_m P
for i ← m − 1 downto 0 do
    if k_i = 0 then
        R ← 2R
    else
        for j ← 0 to w do
            R ← 2R
        end for
        R ← R + k_i P
    end if
end for
return R
```

Analysis of Sliding Window

- We still do $n - 1$ doublings for an n-bit scalar.
- Precomputation needs $2^{w-1} - 1$ additions.
- Expected number of additions in the main loop: $n/(w + 1)$.
- For the same w only half the precomputation compared to fixed-window scalar mult.
- For the same w fewer additions in the main loop.
- But: **It's not running in constant time!**
- Still nice (in double-scalar version) for signature verification.

General Montgomery Ladder

For k in binary represenation with $k = \sum_{i=0}^{n-1} k_i \cdot 2^i$, define

$$L_j = \sum_{i=j}^{n-1} k_i \cdot 2^{i-j}, \quad H_j = L_j + 1$$

We have:

$$\begin{aligned} L_j &= 2L_{j+1} + k_j \\ &= L_{j+1} + H_{j+1} - 1 + k_j \\ &= 2H_{j+1} - 2 + k_j \end{aligned}$$

$$(L_j, H_j) = \begin{cases} (2L_{j+1},\ L_{j+1} + H_{j+1}) \text{ for } k_j = 0 \\ (L_{j+1} + H_{j+1},\ 2H_{j+1}) \text{ for } k_j = 1 \end{cases}$$

$$k = \underbrace{11011}_{L_4=13}011_b = 219$$

$$k = \underbrace{11011}_{L_3=27}011_b = 219$$

$$11011011_b = 219$$

$(1,2)$ $(13,14)$ $(109,110)$
$(3,4)$ $(27,28)$ $(219,220)$
$(6,7)$ $(54,55)$

General Montgomery Ladder

Require: point P, scalar $k = \sum_{i=0}^{n-1} k_i \cdot 2^i$
Ensure: $L = kP$

```
L = 0
H = P
for i = n − 1 downto 0 do
    if k_i = 0 then
        L = point_double(L)
        H = point_add(L, H)
    else
        L = point_add(L, H)
        H = point_double(H)
    end if
end for
return L
```

Department of Mathematics
and Computer Science (IMADA)

- Consider Montgomery curves of the form $by^2 = x^3 + ax^2 + x$.
- Montgomery in 1987 showed how to perform x-coordinate-based arithmetic:
  - Given the x-coordinate x_P of P, and
  - given the x-coordinate x_Q of Q, and
  - given the x-coordinate x_{P-Q} of $P - Q$,
  - compute the x-coordinate x_R of $R = P + Q$.
- This is called differential addition.
- Less efficient differential-addition formulas for other curve shapes.
- Can be used for efficient computation of the x-coordinate of kP given only the x-coordinate of P.
- For this, let's use projective representation $(X : Z)$ with $x = (X/Z)$.

Montgomery ``Ladder Step''

Require: $(X_{Q-P}, X_P, Z_P, X_Q, Z_Q)$
Ensure: $(X_{2P}, Z_{2P}, X_{P+Q}, Z_{P+Q})$
Constant: $a_{24} = (a+2)/4$

$t_1 \leftarrow X_P + Z_P$
$t_6 \leftarrow t_1^2$
$t_2 \leftarrow X_P - Z_P$
$t_7 \leftarrow t_2^2$
$t_5 \leftarrow t_6 - t_7$
$t_3 \leftarrow X_Q + Z_Q$
$t_4 \leftarrow X_Q - Z_Q$
$t_8 \leftarrow t_4 \cdot t_1$
$t_9 \leftarrow t_3 \cdot t_2$
$X_{P+Q} \leftarrow (t_8 + t_9)^2$
$Z_{P+Q} \leftarrow X_{Q-P} \cdot (t_8 - t_9)^2$
$X_{2P} \leftarrow t_6 \cdot t_7$
$Z_{2P} \leftarrow t_5 \cdot (t_7 + a_{24} \cdot t_5)$
return $(X_{2P}, Z_{2P}, X_{P+Q}, Z_{P+Q})$

Require: A scalar $0 \leq k \in \mathbb{Z}$ and the x-coordinate x_P of some point P.
Ensure: (X_{kP}, Z_{kP}) with $x_{kP} = X_{kP}/Z_{kP}$.
$X_1 = x_P; X_2 = 1; Z_2 = 0; X_3 = x_P; Z_3 = 1$

for $i \leftarrow n-1$ **downto** 0 **do**
  if $k_i = 1$ **then**
    $(X_3, Z_3, X_2, Z_2) \leftarrow \text{ladderstep}(X_1, X_3, Z_3, X_2, Z_2)$
  else
    $(X_2, Z_2, X_3, Z_3) \leftarrow \text{ladderstep}(X_1, X_2, Z_2, X_3, Z_3)$
  end if

end for
return X_2/Z_2

Require: A scalar $0 \leq k \in \mathbb{Z}$ and the x-coordinate x_P of some point P.
Ensure: (X_{kP}, Z_{kP}) with $x_{kP} = X_{kP}/Z_{kP}$.

$X_1 = x_P; X_2 = 1; Z_2 = 0; X_3 = x_P; Z_3 = 1$

for $i \leftarrow n-1$ **downto** 0 **do**
  $c \leftarrow k_i$
  $(X_2, X_3) \leftarrow \text{cswap}(X_2, X_3, c)$
  $(Z_2, Z_3) \leftarrow \text{cswap}(Z_2, Z_3, c)$
  $(X_2, Z_2, X_3, Z_3) \leftarrow \text{ladderstep}(X_1, X_2, Z_2, X_3, Z_3)$
  $(X_2, X_3) \leftarrow \text{cswap}(X_2, X_3, c)$
  $(Z_2, Z_3) \leftarrow \text{cswap}(Z_2, Z_3, c)$
end for
return X_2/Z_2

Require: A scalar $0 \leq k \in \mathbb{Z}$ and the x-coordinate x_P of some point P.
Ensure: (X_{kP}, Z_{kP}) with $x_{kP} = X_{kP}/Z_{kP}$.

$X_1 = x_P; X_2 = 1; Z_2 = 0; X_3 = x_P; Z_3 = 1$
$p \leftarrow 0$
for $i \leftarrow n-1$ **downto** 0 **do**
  $b \leftarrow k_i$
  $c \leftarrow b \oplus p$
  $p \leftarrow b$
  $(X_2, X_3) \leftarrow \text{cswap}(X_2, X_3, c)$
  $(Z_2, Z_3) \leftarrow \text{cswap}(Z_2, Z_3, c)$
  $(X_2, Z_2, X_3, Z_3) \leftarrow \text{ladderstep}(X_1, X_2, Z_2, X_3, Z_3)$
end for
return X_2/Z_2

Advantages of the Montgomery Ladder

- Very regular structure, easy to protect against timing attacks.
  - Replace the if statement by conditional swap.
  - Be careful with constant-time swaps.
- Typically very fast.
- Point compression/decompression is free.
- Easy to implement.
- No ugly special cases (see Bernstein's "Curve25519" paper).

Department of Mathematics
and Computer Science (IMADA)

- Consider computation $Q = \sum_1^n k_i P_i$.
- We looked at $n = 2$ before, how about $n = 128$?
- Idea: Assume $k_1 > k_2 > \cdots > k_n$.
- Bos-Coster algorithm: recursively re-order and compute
  $Q = (k_1 - k_2)P_1 + k_2(P_1 + P_2) + k_3 P_3 + \cdots + k_n P_n$,
  $Q = \quad k_1' \quad P_1 + k_2 \quad P_2' \quad + k_3 P_3 + \cdots + k_n P_n$.
- Each step requires one scalar subtraction and one point addition.
- Can be very fast (but not constant-time).
- Requires fast access to the two largest scalars: Put scalars into a heap.
- Crucial for good performance: Fast heap implementation.

How about fixed scalar?

- So far we have considered:
  - **variable** point, **variable** scalar
  - **fixed** point, **variable** scalar
- How about **variable** point, **fixed** scalar?
- Optimizing for the scalar means that the scalar has to be public.
- Not the typical setting for ECC.
- Some applications:
  - Inversion in finite fields.
  - Elliptic-curve factorization method (not in this lecture).

Definition:
Let k be a positive integer. A sequence $s_1, s_2, \ldots, s_m$ is called an addition chain of length m for k if:

- $s_1 = 1$,
- $s_m = k$, and
- for each s_i with $i > 1$ it holds that $s_i = s_j + s_k$ for some $j, k < i$.

An addition chain for k immediately translates into a scalar multiplication algorithm to compute kP:

- Start with $s_1 P = P$.
- Compute $s_i P = s_j P + s_k P$ for $i = 2, \ldots, m$.

- All algorithms so far just computed additions chains “on the fly”.
- Signed-scalar representations are “addition-subtraction chains”.
- For a fixed scalar, we can spend a lot of time to find a good addition chain at compile time.
- This is what we used for inversion in $\mathbb{F}_{2^{255}-19}$.
- Get them for free:
  `wwwhomes.uni-bielefeld.de/achim/addition_chain.html`