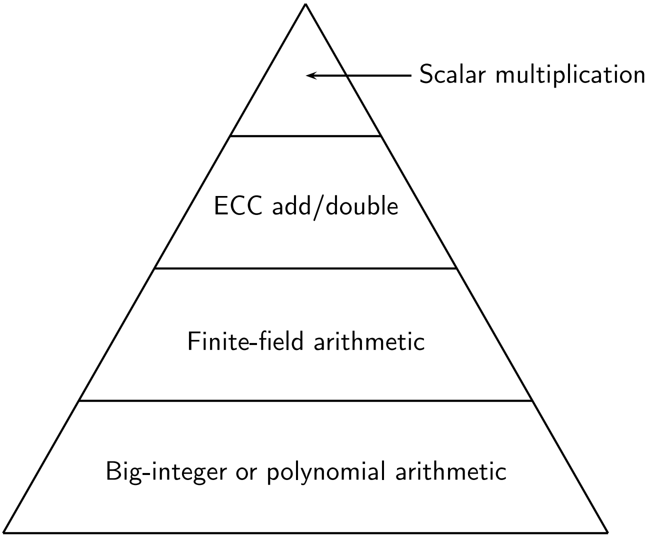
Cryptographic Engineering

Scalar multiplication

Radboud University, Nijmegen, The Netherlands

Spring 2019

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 G, written additively
  - Compute $k \cdot P$ for $k \in \mathbb{Z}$ and $P \in G$
  - This is the same as x^k for x in a multiplicative group G'
  - Same algorithms for scalar multiplication and exponentiation

The ECDLP

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

EC Diffie-Hellman key exchange

- Users Alice and Bob have key pairs (k_A, Q_A) and (k_B, Q_B)
- Alice sends Q_A to Bob
- Bob sends Q_B to Alice
- Alice computes joint key as $K = k_A Q_B$
- Bob computes joint key as $K = k_B Q_A$

Schnorr signatures

- Alice has key pair (k_A, Q_A)
- Order of $\langle P \rangle$ is ℓ
- Use cryptographic hash function H
- Sign: Generate secret random $r \in \{1, \ldots, \ell\}$, compute signature $(H(R, M), S)$ on M with

$$R = rP$$
$$S = (r - H(R, M)k_A) \mod \ell$$

- Verify: compute $\overline{R} = SP + H(R, M)Q_A$ and check that

$$H(\overline{R}, M) = H(R, M)$$

Scalar multiplication

- 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

A first approach

- Let's compute $105 \cdot P$.
- Obvious: Can do that with 104 additions $P + P + P + \cdots + P$
- Problem: 105 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)

Rewriting the scalar

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$
- $105 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
- $105 = (((((((((1 \cdot 2 + 1) \cdot 2) + 0) \cdot 2) + 1) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + 1$ (Horner's rule)
- $105 \cdot P = (((((((((P \cdot 2 + P) \cdot 2) + 0) \cdot 2) + P) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + P$
- Cost: 6 doublings, 3 additions
- General algorithm: "Double and add"

  $R \leftarrow P$
  for $i \leftarrow n - 2$ downto 0 **do**
      $R \leftarrow 2R$
      if $(k)_2[i] = 1$ **then**
          $R \leftarrow R + P$
      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: $\approx n/2$ 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: insecure for secret scalars!

Double-scalar double-and-add

- Let's modify the algorithm to compute $k_1 P_1 + k_2 P_2$
- Obvious solution:
  - Compute $k_1 P_1$ ($n_1 - 1$ doublings, $m_1 - 1$ additions)
  - Compute $k_2 P_2$ ($n_2 - 1$ doublings, $m_2 - 1$ additions)
  - Add the results (1 addition)
- We can do better ($\mathcal{O}$ denotes the neutral element):

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

- $\max(n_1, n_2)$ doublings, $m_1 + m_2$ additions

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):

```
R ← O
for i ← max(n1, n2) − 1 downto 0 do
    R ← 2R
    if (k1)2[i] = 1 AND (k2)2[i] = 1 then
        R ← R + T
    else
        if (k1)2[i] = 1 then
            R ← R + P1
        end if
        if (k2)2[i] = 1 then
            R ← R + P2
        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 $0P, P, 2P, 3P, \dots$, when we receive k, simply look up kP.
- Problem: k is large. For a 256-bit k we would need a table of size $3369993333393829974333376885877453834204643052817571560137951281152$TB
- 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:

  $R \leftarrow \mathcal{O}$
  for $i \leftarrow 0$ to $n-1$ **do**
      if $(k)_2[i] = 1$ **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:

  $R \leftarrow P$
  for $i \leftarrow n-2$ downto 0 **do**
      $R \leftarrow 2R$
      $R_t \leftarrow R + P$
      if $(k)_2[i] = 1$ **then**
          $R \leftarrow R_t$
      end if
  end for

- Or simply add the neutral element $\mathcal{O}$

  $R \leftarrow P$
  for $i \leftarrow n-2$ downto 0 **do**
      $R \leftarrow 2R$
      if $(k)_2[i] = 1$ **then**
          $R \leftarrow R + P$
      else
          $R \leftarrow R + \mathcal{O}$

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

  $R \leftarrow P$
  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 $i \leftarrow n-2$ downto 0 **do**
  $\quad R \leftarrow 3R$
  $\quad R \leftarrow R + T[(k)_3[i]]$
  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 length w
- Compute scalar multiplication as

  $R \leftarrow T[(k)_{2^w}[m-1]]$
  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

Analysis of fixed window

- For an n-bit scalar we still have $n-1$ doublings
- Precomputation costs us $2^w/2-1$ 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

```
/* Sets r to the neutral element on the elliptic curve */
extern ec_point_setneutral(ec_point *r);

/* Adds p and q and stores the result in r */
extern ec_point_add(ec_point *r, const ec_point *p, const ec_point *q);

/* Doubles p and stores the result in r */
extern ec_point_double(ec_point *r, const ec_point *p);

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

ec_scalarmult_P(unsigned char scalar[32])
{
  int i,j;
  ec_point r;
  ec_setneutral(&r);

  for(i=31;i>=0;i--)
  {
    for(j=0;j<8;j++)
      ec_point_double(&r,&r);
    if(scalar[i] != 0)
      ec_point_add(&r,&r,precomputed+scalar[i]-1);
  }
}
```

Making it constant time

```
/* Sets r to the neutral element on the elliptic curve */
extern ec_point_setneutral(ec_point *r);

/* Adds p and q and stores the result in r */
extern ec_point_add(ec_point *r, const ec_point *p, const ec_point *q);

/* Doubles p and stores the result in r */
extern ec_point_double(ec_point *r, const ec_point *p);

/* For point P contains pre-computed multiples 0, P, 2*P, 3*P,...,255*P */
extern ec_point precomputed[256];

ec_scalarmult_P(unsigned char scalar[32])
{
  int i,j;
  ec_point r;
  ec_setneutral(&r);

  for(i=31;i>=0;i--)
  {
    for(j=0;j<8;j++)
      ec_point_double(&r,&r);
    ec_point_add(&r,&r,precomputed+scalar[i]);
  }
}
```

Making it constant time

```
/* Sets r to the neutral element on the elliptic curve */
extern ec_point_setneutral(ec_point *r);

/* Adds p and q and stores the result in r */
extern ec_point_add(ec_point *r, const ec_point *p, const ec_point *q);

/* Doubles p and stores the result in r */
extern ec_point_double(ec_point *r, const ec_point *p);

/* For point P contains pre-computed multiples 0, P, 2*P, 3*P,...,255*P */
extern ec_point precomputed[256];

ec_scalarmult_P(unsigned char scalar[32])
{
  int i,j;
  ec_point r,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);
  }
}
```

ec_point_lookup

```
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 = (i == pos); // Not constant time!
    ec_point_cmov(t, table+i, b); // Copy table[i] to t if b is 1
  }
}
```

ec_point_lookup

```
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 $P, 2P, 4P, 8P, \dots$
- We can combine that with fixed-window scalar multiplication
- Precompute $T_i = (\mathcal{O}, P, 2P, 3P, \dots, (2^w - 1)P) \cdot 2^i$ 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**
  $\quad R \leftarrow R + T_{iw}[(k)_{2^w}[i]]$
  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.
  - constant-time loads get slow for large w

Fixed-window limitations

- Consider the scalar $22 = (1\,01\,10)_2$ and window size 2
  - Initialize R with P
  - Double, double, add P
  - Double, double, add $2P$
- More efficient:
  - Initialize R with P
  - Double, double, double, add $3P$
  - double
- Problem with fixed window: it's fixed.
- Idea: "Slide" the window over the scalar

Sliding window scalar multiplication

- Choose window size w
- Rewrite scalar k as $k = (k_0, \dots, k_m)$ with k_i in $\{0, 1, 3, 5, \dots, 2^w - 1\}$ with at most one non-zero entry in each window of length w
- Do this by scanning k from right to left, expand window from each 1-bit
- Precompute $P, 3P, 5P, \dots, (2^w - 1)P$
- Perform scalar multiplication

$R \leftarrow \mathcal{O}$
for $i \leftarrow m$ to 0 **do**
    $R \leftarrow 2R$
    if k_i **then**
        $R \leftarrow R + k_i P$
    end if
end for

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 multiplication
- 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

Differential addition

- Consider elliptic 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 $x = (X/Z)$

One Montgomery "ladder step"

const $a24 = (A+2)/4$ (A from the curve equation)
function ladderstep($X_{Q-P}, X_P, Z_P, X_Q, Z_Q$)
  $t_1 \leftarrow X_P + Z_P$
  $t_6 \leftarrow t_1^2$
  $t_2 \leftarrow X_P - Z_P$
  $t_7 \leftarrow t_2^2$
  $t_5 \leftarrow t_6 - t_7$
  $t_3 \leftarrow X_Q + Z_Q$
  $t_4 \leftarrow X_Q - Z_Q$
  $t_8 \leftarrow t_4 \cdot t_1$
  $t_9 \leftarrow t_3 \cdot t_2$
  $X_{P+Q} \leftarrow (t_8 + t_9)^2$
  $Z_{P+Q} \leftarrow X_{Q-P} \cdot (t_8 - t_9)^2$
  $X_{2P} \leftarrow t_6 \cdot t_7$
  $Z_{2P} \leftarrow t_5 \cdot (t_7 + a24 \cdot t_5)$
  return $(X_{2P}, Z_{2P}, X_{P+Q}, Z_{P+Q})$
end function

The Montgomery ladder

Require: A scalar $0 \leq k \in \mathbb{Z}$ and the x-coordinate x_P of some point P
Ensure: (X_{kP}, Z_{kP}) fulfilling $x_{kP} = X_{kP}/Z_{kP}$
$X_1 = x_P$; $X_2 = 1$; $Z_2 = 0$; $X_3 = x_P$; $Z_3 = 1$
for $i \leftarrow n - 1$ downto 0 **do**
  if bit i of k is 1 **then**
    $(X3, Z3, X2, Z2) \leftarrow$ ladderstep($X1, X3, Z3, X2, Z2$)
  else
    $(X2, Z2, X3, Z3) \leftarrow$ ladderstep($X1, X2, Z2, X3, Z3$)
  end if
end for
return X_2/Z_2

The Montgomery ladder (ctd.)

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

$X_1 = x_P$; $X_2 = 1$; $Z_2 = 0$; $X_3 = x_P$; $Z_3 = 1$
for $i \leftarrow n - 1$ downto 0 **do**
  $b \leftarrow$ bit i of s
  $c \leftarrow b \oplus p$
  $p \leftarrow b$
  $(X2, X3) \leftarrow$ cswap$(X2, X3, c)$
  $(Z2, Z3) \leftarrow$ cswap$(Z2, Z3, c)$
  $(X2, Z2, X3, Z3) \leftarrow$ ladderstep$(X1, X2, Z2, X3, Z3)$
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
- Very fast (at least if we don't compare to curves with efficient endomorphisms)
- Point compression/decompression is free
- Easy to implement
- No ugly special cases (see Bernstein's "Curve25519" paper)

Multi-scalar multiplication

- 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 compute $Q = (k_1 - k_2)P_1 + k_2(P_1 + P_2) + k_3P_3 \cdots + k_nP_n$
- 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

A fast heap

- Heap is a binary tree, each parent node is larger than the two child nodes
- Data structure is stored as a simple array, positions in the array determine positions in the tree
- Root is at position 0, left child node at position 1, right child node at position 2 etc.
- For node at position i, child nodes are at position $2 \cdot i + 1$ and $2 \cdot i + 2$, parent node is at position $\lfloor (i - 1)/2 \rfloor$
- Typical heap root replacement (pop operation): start at the root, swap down for a variable amount of times
- Floyd's heap: swap down to the bottom, swap up for a variable amount of times, advantages:
  - Each swap-down step needs only one comparison (instead of two)
  - Swap-down loop is more friendly to branch predictors

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 (cmp. slides 17&18 of ecc.pdf)
  - Elliptic-curve factorization method (not in this lecture)

Addition chains

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

- $s_1 = 1$
- $s_m = k$
- 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 fixed scalar we can spend a lot of time to find a good addition chain at compile time
- This is what was used for inversion in $\mathbb{F}_{2^{255}-19}$