Scalar-multiplication algorithms

Peter Schwabe
Radboud University Nijmegen, The Netherlands

September 11, 2013

ECC 2013 Summer School
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 $kP = Q$. 
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 \((k_B, Q_B)\)

1. Alice sends \(Q_A\) to Bob
2. Bob sends \(Q_B\) to Alice
3. Alice computes joint key as \(K = k_A Q_B\)
4. Bob computes joint key as \(K = k_B Q_A\)
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
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 \(\ell\)
- Use cryptographic hash function \(H\)
Schnorr signatures

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

\[
R = rP \\
S = (r + H(R, M)k_A) \mod \ell
\]
Schnorr signatures

- Alice has key pair \((k_A, Q_A)\)
- Order of \(\langle P \rangle\) is \(\ell\)
- 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 \(\bar{R} = SP + H(R, M)Q_A\) and check that
  \[
  H(\bar{R}, M) = H(R, M)
  \]
Scalar multiplication

- Looks like all these schemes need computation of $kP$. 
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

Schnorr signature verification needs double-scalar multiplication $k_1P_1 + k_2P_2$.
Scalar multiplication

- Looks like all these schemes need computation of \( kP \).
- Let’s take a closer look:
  - For key generation, the point \( P \) is \textit{fixed} at compile time
  - For Diffie-Hellman joint-key computation the point is received at runtime
  - Key generation and Diffie-Hellman need \textit{one} scalar multiplication \( kP \)
  - Schnorr signature verification needs double-scalar multiplication
    \[ k_1 P_1 + k_2 P_2 \]
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 $k_1 P_1 + k_2 P_2$
  - In key generation and Diffie-Hellman joint-key computation, $k$ is secret
  - The scalars in Schnorr signature verification are public
Scalar multiplication

- Looks like all these schemes need computation of $kP$.
- Let’s take a closer look:
  - For key generation, the point $P$ is \textit{fixed} at compile time
  - For Diffie-Hellman joint-key computation the point is received at runtime
  - Key generation and Diffie-Hellman need \textit{one} scalar multiplication $kP$
  - Schnorr signature verification needs double-scalar multiplication $k_1 P_1 + k_2 P_2$
  - 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
The computation $kP$ should have the same result for public or for secret $k$. 

Problem: Timing information:

Some fast scalar-multiplication algorithms have a running time that depends on $k$.

An attacker can measure time and deduce information about $k$.

Brumley, Tuveri, 2011: A few minutes to steal the private key of a TLS server over the network.
Secret vs. public scalars

- The computation \( kP \) should have the same result for public or for secret \( k \)
- True. We still want different algorithms.
- Problem: Timing information:
  - Some fast scalar-multiplication algorithms have a running time that depends on \( k \)
  - An attacker can measure time and deduce information about \( k \)
Secret vs. public scalars

- The computation $kP$ should have the same result for public or for secret $k$

- True. We still want different algorithms.

- Problem: Timing information:
  - Some fast scalar-multiplication algorithms have a running time that depends on $k$
  - An attacker can measure time and deduce information about $k$
  - Brumley, Tuveri, 2011: A few minutes to steal the private key of a TLS server over the network.
Secret vs. public scalars

The computation $kP$ should have the same result for public or for secret $k$

True. We still want different algorithms.

Problem: Timing information:

- Some fast scalar-multiplication algorithms have a running time that depends on $k$
- An attacker can measure time and deduce information about $k$
- Brumley, Tuveri, 2011: A few minutes to steal the private key of a TLS server over the network.
- For secret $k$ we need constant-time algorithms
A first approach

- Let’s compute $105 \cdot P$. 

Obvious: Can do that with $10^4$ additions $P + P + P + \cdots + P$

Problem: $105$ has $7$ bits, we need roughly $2^7$ additions, real scalars have $\approx 256$ 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)
A first approach

- Let’s compute $105 \cdot P$.
- Obvious: Can do that with 104 additions $P + P + P + \cdots + P$
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, *real* scalars have $\approx 256$ bits, we would need roughly $2^{256}$ additions (more expensive than solving the ECDLP!)
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, real scalars have $\approx 256$ 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

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$
Rewriting the scalar

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$
- $105 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
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)
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)
- \[ 105 \cdot P = (((((((((P \cdot 2 + P) \cdot 2) + 0) \cdot 2) + P) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + P \]
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
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
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: $\approx \frac{n}{2}$ additions
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 $P$
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
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 $k_1 P_1 + k_2 P_2$
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 ($O$ denotes the neutral element):

$$R \leftarrow O$$

$$\text{for } i \leftarrow \max(n_1, n_2) - 1 \text{ downto } 0$$

$$R \leftarrow 2R$$

\begin{align*}
  &\text{if } (k_1)_2[i] = 1 \\
  &\text{then } R \leftarrow R + P_1
\end{align*}$$

\begin{align*}
  &\text{if } (k_2)_2[i] = 1 \\
  &\text{then } R \leftarrow R + P_2
\end{align*}$$

end for

return $R$
Let’s modify the algorithm to compute $k_1P_1 + k_2P_2$

Obvious solution:

- Compute $k_1P_1$ ($n_1 - 1$ doublings, $m_1 - 1$ additions)
- Compute $k_2P_2$ ($n_2 - 1$ doublings, $m_2 - 1$ additions)
- Add the results (1 addition)

We can do better ($\mathcal{O}$ denotes the neutral element):

```plaintext
R ← O
for i ← max($n_1, n_2$) − 1 downto 0 do
    R ← 2R
    if $(k_1)_2[i] = 1$ then
        R ← R + $P_1$
    end if
    if $(k_2)_2[i] = 1$ then
        R ← R + $P_2$
    end if
end for
return R
```
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 ($O$ denotes the neutral element):
  
  ```
  R ← O
  for $i$ ← max($n_1, n_2$) − 1 downto 0 do
    $R$ ← $2R$
    if $(k_1)_2[i] = 1$ then
      $R$ ← $R + P_1$
    end if
    if $(k_2)_2[i] = 1$ then
      $R$ ← $R + P_2$
    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)
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 \(T = P_1 + P_2\)
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):

\[
R \leftarrow O \\
\text{for } i \leftarrow \max(n_1, n_2) - 1 \text{ downto } 0 \text{ do} \\
\quad R \leftarrow 2R \\
\quad \text{if } (k_1)_2[i] = 1 \text{ AND } (k_2)_2[i] = 1 \text{ then} \\
\quad \quad R \leftarrow R + T \\
\quad \text{else} \\
\quad \quad \text{if } (k_1)_2[i] = 1 \text{ then} \\
\quad \quad \quad R \leftarrow R + P_1 \\
\quad \quad \text{end if} \\
\quad \quad \text{if } (k_2)_2[i] = 1 \text{ then} \\
\quad \quad \quad R \leftarrow R + P_2 \\
\quad \quad \text{end if} \\
\quad \text{end if} \\
\text{end for} \\
\text{return } R
\]
Even more (offline) precomputation

- What if precomputation is free (fixed basepoint, offline precomputation)?

  - First idea: Let’s precompute a table containing $P, P^2, P^3, ..., P^k$, 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 $P, P^2, P^4, P^8, ..., P^{2^n-1}$?

  - This needs only about 8 KB of storage for $n = 256$.

  - Modified scalar-multiplication algorithm:

    ```plaintext
    R ← 0
    for i ← 0 to n−1 do
      if (k)2[i] = 1 then
        R ← R + 2iP
      end if
    end for
    return R
    ```

  - Eliminated all doublings in fixed-basepoint scalar multiplication!
Even more (offline) precomputation

- What if precomputation is free (fixed basepoint, offline precomputation)?
- First idea: Let’s precompute a table containing $0P, P, 2P, 3P, \ldots$, when we receive $k$, simply look up $kP$.
Even more (offline) precomputation

- What if precomputation is free (fixed basepoint, offline precomputation)?
- First idea: Let’s precompute a table containing $0P, P, 2P, 3P, \ldots$, when we receive $k$, simply look up $kP$.
- Problem: $k$ is large. For a 256-bit $k$ we would need a table of size 
  \[3369993333393829974333376885877453834204643052817571560137951281152\text{TB}\]
Even more (offline) precomputation

- What if precomputation is free (fixed basepoint, offline precomputation)?
- First idea: Let’s precompute a table containing $0P, P, 2P, 3P, \ldots$, 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 $P, 2P, 4P, 8P, \ldots, 2^{n-1}P$
- This needs only about 8KB of storage for $n = 256$
Even more (offline) precomputation

- What if precomputation is free (fixed basepoint, offline precomputation)?
- First idea: Let’s precompute a table containing $0P, P, 2P, 3P, \ldots$, when we receive $k$, simply look up $kP$.
- Problem: $k$ is large. For a 256-bit $k$ we would need a table of size $33699933333393829974333376885877453834204643052817571560137951281152$ TB.
- How about, for example, precompute $P, 2P, 4P, 8P, \ldots, 2^{n-1}P$.
- This needs only about 8KB of storage for $n = 256$.
- Modified scalar-multiplication algorithm:

```plaintext
R ← O
for i ← 0 to \( n - 1 \) do
    if \((k)_2[i] = 1\) then
        \( R ← R + 2^iP \)
    end if
end for
return R
```

Eliminated all doublings in fixed-basepoint scalar multiplication!
Even more (offline) precomputation

- What if precomputation is free (fixed basepoint, offline precomputation)?
- First idea: Let’s precompute a table containing $0P, P, 2P, 3P, \ldots$, 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 $P, 2P, 4P, 8P, \ldots$, $2^{n-1}P$
- This needs only about 8KB of storage for $n = 256$
- Modified scalar-multiplication algorithm:

  \[
  R \leftarrow O \\
  \text{for } i \leftarrow 0 \text{ to } n - 1 \text{ do} \\
  \quad \text{if } (k)_2[i] = 1 \text{ then} \\
  \qquad R \leftarrow R + 2^i P \\
  \text{end if} \\
  \text{end for} \\
  \text{return } R
  \]

- 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
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:

\[
\begin{align*}
R & \leftarrow P \\
\text{for } i \leftarrow n - 2 \text{ downto } 0 & \text{ do} \\
& \quad R \leftarrow 2R \\
& \quad R_t \leftarrow R + P \\
& \quad \text{if } (k)_2[i] = 1 \text{ then} \\
& \quad & \quad R \leftarrow R_t \\
& \quad \text{end if} \\
& \text{end for}
\end{align*}
\]

- Or simply add the neutral element \(O\)
- Still not constant time, more later...

Scalar-multiplication algorithms
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 $\mathcal{O}$

\[
\begin{align*}
R & \leftarrow P \\
\text{for } i & \leftarrow n - 2 \ 	ext{down to } 0 \ 	ext{do} \\
R & \leftarrow 2R \\
\text{if } (k)_2[i] = 1 \ 	ext{then} \\
R & \leftarrow R + P \\
\text{else} \\
R & \leftarrow R + \mathcal{O}
\end{align*}
\]

end if
end for
return $R$

Still not constant time, more later...
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 \( O \)

\[
\begin{align*}
R &\leftarrow P \\
\text{for } i &\leftarrow n - 2 \text{ downto } 0 \text{ do} \\
R &\leftarrow 2R \\
\text{if } (k)_2[i] = 1 \text{ then} \\
R &\leftarrow R + P \\
\text{else} \\
R &\leftarrow R + O \\
\text{end if} \\
\text{end for} \\
\text{return } R
\end{align*}
\]

- Still not constant time, more later...
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

\[
\begin{align*}
R & \leftarrow P \\
\text{for } i & \leftarrow n - 2 \text{ downto } 0 \text{ do} \\
R & \leftarrow 2R \\
R & \leftarrow R + T[(k)_2[i]]
\end{align*}
\]

end for
Changing the scalar radix

- So far we considered a scalar written in radix 2
- How about radix 3?

We precompute a Table $T = (O, P, 2P)$

Write scalar $k$ as $(k_{n-1}, ..., k_0)_3$

Compute scalar multiplication as $R ← T[(k)_3[n-1]]$ for $i ← n-2$ downto $0$
do

$R ← 3R$

$R ← R + T[(k)_3[i]]$

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?
Changing the scalar radix

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

- So far we considered a scalar written in radix 2
- How about radix 3?
- We precompute a Table $T = (O, P, 2P)$
- Write scalar $k$ as $(k_{n-1}, \ldots, k_0)_3$
- Compute scalar multiplication as
  \[
  R \leftarrow T[(k)_3[n - 1]] \\
  \text{for } i \leftarrow n - 2 \text{ downto } 0 \text{ do} \\
  \quad R \leftarrow 3R \\
  \quad R \leftarrow R + T[(k)_3[i]] \\
  \text{end for}
  \]

Advantage: The scalar is shorter, fewer additions
Disadvantage: 3 is just not nice (needs triplings)
How about some nice numbers, like 4, 8, 16?
Changing the scalar radix

- So far we considered a scalar written in radix 2
- How about radix 3?
- We precompute a Table $T = (O, P, 2P)$
- Write scalar $k$ as $(k_{n-1}, \ldots, k_0)_3$
- Compute scalar multiplication as
  \[
  R \leftarrow T[(k)_3[n-1]] \\
  \text{for } i \leftarrow n-2 \text{ downto } 0 \text{ do} \\
  \quad R \leftarrow 3R \\
  \quad R \leftarrow R + T[(k)_3[i]]
  \]
  \text{end for}
- Advantage: The scalar is shorter, fewer additions
- Disadvantage: 3 is just not nice (needs triplings)
Changing the scalar radix

- So far we considered a scalar written in radix $2$
- How about radix $3$?
- We precompute a Table $T = (O, P, 2P)$
- 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
  $R \leftarrow 3R$
  $R \leftarrow R + T[(k)_3[i]]$
  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 $T = (O, P, 2P, \ldots, (2^w - 1)P)$
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 \( w \)
Fixed-window scalar multiplication

- Fix a window width $w$
- Precompute $T = (O, P, 2P, \ldots, (2^w - 1)P)$
- 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
Analysis of fixed window

- For an $n$-bit scalar we still have $n - 1$ doublings
- Precomputation costs us $w/2 - 1$ additions and $w/2 - 1$ doublings
Analysis of fixed window

- For an $n$-bit scalar we still have $n - 1$ doublings
- Precomputation costs us $w/2 - 1$ additions and $w/2 - 1$ doublings
- Number of additions in the loop is $\lceil n/w \rceil$
Analysis of fixed window

- For an $n$-bit scalar we still have $n - 1$ doublings
- Precomputation costs us $w/2 - 1$ additions and $w/2 - 1$ doublings
- Number of additions in the loop is $\lceil n/w \rceil$
- Larger $w$: More precomputation
- Smaller $w$: More additions inside the loop

For $\approx 256$-bit scalars choose $w = 4$ or $w = 5$
Analysis of fixed window

- For an $n$-bit scalar we still have $n - 1$ doublings
- Precomputation costs us $w/2 - 1$ additions and $w/2 - 1$ doublings
- Number of additions in the loop is $\lceil n/w \rceil$
- Larger $w$: More precomputation
- Smaller $w$: More additions inside the loop
- For $\approx 256$-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.
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 (hint: you want to use Edward’s curves)
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 $O$?
  - We can make that work, but how easy and efficient it is depends on the curve shape (hint: you want to use Edward’s curves)
  - Are lookups from the table $T$ running in constant time?
  - Usually not!
Cache-timing attacks

- We load from table $T$ at position $p = (k)_{2^w}[i]$
- The position is part of the secret scalar, so also secret
Cache-timing attacks

- We load from table $T$ at position $p = (k)_{2w}[i]$
- The position is part of the secret scalar, so also secret
- Most processors load data through several caches (transparent, fast memory)
  - loads are fast if data is found in cache (cache hit)
  - loads are slow if data is not found in cache (cache miss)

Problem 1: if-statements are not constant time
Problem 2: Comparisons are not (guaranteed to be) constant time
Cache-timing attacks

- We load from table $T$ at position $p = (k_i)_{2^w}[i]
- The position is part of the secret scalar, so also secret
- Most processors load data through several caches (transparent, fast memory)
  - loads are fast if data is found in cache (cache hit)
  - loads are slow if data is not found in cache (cache miss)
- Solution (part 1): Load all items, pick the right one:

\[
R \leftarrow O \\
\text{for } i \text{ from 1 to } 2^w - 1 \text{ do} \\
\quad \text{if } p = i \text{ then} \\
\quad \quad R \leftarrow T[i] \\
\text{end if} \\
\text{end for}
\]
Cache-timing attacks

- We load from table $T$ at position $p = (k)_{2w}[i]$
- The position is part of the secret scalar, so also secret
- Most processors load data through several caches (transparent, fast memory)
  - loads are fast if data is found in cache (cache hit)
  - loads are slow if data is not found in cache (cache miss)
- Solution (part 1): Load all items, pick the right one:
  
  $R \leftarrow O$
  
  for $i$ from 1 to $2^w - 1$
  
  if $p = i$ then
  
  $R \leftarrow T[i]$
  
  end if
  
  end for

- Problem 1: if-statements are not constant time
Cache-timing attacks

- We load from table $T$ at position $p = (k^i)_{2^w}$
- The position is part of the secret scalar, so also secret
- Most processors load data through several caches (transparent, fast memory)
  - loads are fast if data is found in cache (cache hit)
  - loads are slow if data is not found in cache (cache miss)
- Solution (part 1): Load all items, pick the right one:
  
  $R \leftarrow O$
  
  for $i$ from 1 to $2^w - 1$
  
  if $p = i$ then
    
    $R \leftarrow T[i]$
  
  end if

  end for

- Problem 1: if-statements are not constant time
- Problem 2: Comparisons are not (guaranteed to be) constant time
Constant-time ifs

- A general if statement looks as follows:
  ```
  if s then
      R ← A
  else
      R ← B
  end if
  ```
- This takes different amount of time depending on the bit \( s \), even if \( A \) and \( B \) take the same amount of time.
- Reason: branch prediction
Constant-time ifs

- A general if statement looks as follows:
  ```
  if s then
    R ← A
  else
    R ← B
  end if
  ```
- This takes different amount of time depending on the bit $s$, even if $A$ and $B$ take the same amount of time.
- Reason: branch prediction
- Suitable replacement:
  $$R ← s \cdot A + (1 - s) \cdot B$$
Constant-time ifs

- A general if statement looks as follows:
  
  ```
  if s then
      R ← A
  else
      R ← B
  end if
  ```

- This takes different amount of time depending on the bit $s$, even if $A$ and $B$ take the same amount of time.

- Reason: branch prediction

- Suitable replacement:
  
  ```
  R ← s \cdot A + (1 - s) \cdot B
  ```

- Can replace multiplication and addition with bit-logical operations (AND and XOR)
Constant-time ifs

- A general if statement looks as follows:
  
  ```
  if s then
    R ← A
  else
    R ← B
  end if
  ```

- This takes different amount of time depending on the bit $s$, even if $A$ and $B$ take the same amount of time.

- Reason: branch prediction

- Suitable replacement:
  
  $$R ← s \cdot A + (1 - s) \cdot B$$

- Can replace multiplication and addition with bit-logical operations (AND and XOR)

- For very fast $A$ and $B$, this can even be faster than the conditional branch
Constant-time comparison

static unsigned long long eq(unsigned char a, unsigned char b)
{
    unsigned long long t = a ^ b;
    t = (-t) >> 63;
    return 1-t;
}
More offline precomputation

- Let’s get back to fixed-basepoint multiplication
- So far we precomputed $P, 2P, 4P, 8P, \ldots$
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 \( T_i = (\emptyset, P, 2P, 3P, \ldots, (2^w - 1)P) \cdot 2^i \) for 
  \( i = 0, w, 2w, 3w, \lceil n/w \rceil - 1 \)
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 \( T_i = (O, P, 2P, 3P, \ldots, (2^w - 1)P) \cdot 2^i \) for \( i = 0, w, 2w, 3w, \lceil n/w \rceil - 1 \)
- Perform scalar multiplication as
  \[
  R \leftarrow T_0[(k)_{2^w}[0]]
  \]
  \[
  \text{for } i \leftarrow 1 \text{ to } \lceil n/w \rceil - 1 \text{ do}
  \]
  \[
  R \leftarrow R + T_i[(k)_{2^w}[i]]
  \]
  \[
  \text{end for}
  \]
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 $T_i = (O, P, 2P, 3P, \ldots, (2^w - 1)P) \cdot 2^i$ for $i = 0, w, 2w, 3w, \lceil n/w \rceil - 1$
- Perform scalar multiplication as
  
  $$R \leftarrow T_0[(k)_{2^w}]$$
  
  for $i \leftarrow 1$ to $\lceil n/w \rceil - 1$ do
  
  $$R \leftarrow R + T_i[(k)_{2^w}]$$

  end for

- No doublings, only $\lceil b/w \rceil - 1$ additions
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 $T_i = (O, P, 2P, 3P, \ldots, (2^w - 1)P) \cdot 2^i$ for $i = 0, w, 2w, 3w, \lceil n/w \rceil - 1$
- Perform scalar multiplication as
  
  $R \leftarrow T_0[(k)_{2w}[0]]$
  for $i \leftarrow 1$ to $\lceil n/w \rceil - 1$ do
  
  $R \leftarrow R + T_i[(k)_{2w}[i]]$

- No doublings, only $\lceil b/w \rceil - 1$ 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 $22 = (10110)_2$ and window size 2
  - Initialize $R$ with $P$
  - Double, double, add $P$
  - Double, double, add $2P$
Fixed-window limitations

- Consider the scalar $22 = (10110)_2$ 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
Fixed-window limitations

- Consider the scalar $2^2 = (10110)_2$ 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.
Fixed-window limitations

- Consider the scalar $2^2 = (10110)_2$ 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, 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 $k = (k_0, \ldots, k_m)$ with $k_i$ in \{0, 1, 3, 5, \ldots, 2^w - 1\} with at most one non-zero entry in each window of length $w$
Sliding window scalar multiplication

- Choose window size $w$
- Rewrite scalar $k$ as $k = (k_0, \ldots, k_m)$ with $k_i$ in
  $\{0, 1, 3, 5, \ldots, 2^w - 1\}$ 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
Sliding window scalar multiplication

- Choose window size $w$
- Rewrite scalar $k$ as $k = (k_0, \ldots, k_m)$ with $k_i$ in
  \(\{0, 1, 3, 5, \ldots, 2^w - 1\}\) 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 $P, 3P, 5P, \ldots, (2^w - 1)P$
Sliding window scalar multiplication

- Choose window size $w$
- Rewrite scalar $k$ as $k = (k_0, \ldots, k_m)$ with $k_i$ in
  \{0, 1, 3, 5, \ldots, 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 $P, 3P, 5P, \ldots, (2^w - 1)P$
- Perform scalar multiplication
  
  \[
  R \leftarrow O
  \]
  
  for $i \leftarrow m$ to 0 do
    \[
    R \leftarrow 2R
    \]
    if $k_i$ then
      \[
      R \leftarrow R + k_iP
      \]
    end if
  end for
Analysis of sliding window

- We still do $n - 1$ doublings for an $n$-bit scalar
- Precomputation needs $2^{w-1}$
- Expected number of additions in the main loop: $n/(w + 1)$
Analysis of sliding window

- We still do $n - 1$ doublings for an $n$-bit scalar
- Precomputation needs $2^{w-1}$
- 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
Analysis of sliding window

- We still do \( n - 1 \) doublings for an \( n \)-bit scalar
- Precomputation needs \( 2^{w-1} \)
- 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
Using efficient negation

- So far everything we did works for any cyclic group $\langle P \rangle$
- Elliptic curves have so much more to offer
- For example, efficient negation: $-(x, y) = (x, -y)$ (on Weierstrass curves)
Using efficient negation

- So far everything we did works for any cyclic group $\langle P \rangle$.
- Elliptic curves have so much more to offer.
- For example, efficient negation: $-(x, y) = (x, -y)$ (on Weierstrass curves).
- Idea: use a signed representation for the scalar.
- Fixed-window scalar multiplication:
  - Write scalar as $(k_0, \ldots, k_{m-1})$ with $k_i \in [-2^w, \ldots, 2^w - 1]$.
  - Precompute $T = (-2^w P, (-2^w + 1)P, \ldots, \mathcal{O}, P, \ldots, (2^w - 1)P)$.
  - Perform normal fixed-window scalar multiplication.
  - Half of the precomputation is almost free, we get one bit of $w$ for free.
Using efficient negation

- So far everything we did works for any cyclic group $\langle P \rangle$
- Elliptic curves have so much more to offer
- For example, efficient negation: $-(x, y) = (x, -y)$ (on Weierstrass curves)
- Idea: use a signed representation for the scalar
- Fixed-window scalar multiplication:
  - Write scalar as $(k_0, \ldots, k_{m-1})$ with $k_i \in [-2^w, \ldots, 2^w - 1]$
  - Precompute $T = (-2^w P, (-2^w + 1) P, \ldots, \mathcal{O}, P, \ldots, (2^w - 1) P$
  - Perform normal fixed-window scalar multiplication
  - Half of the precomputation is almost free, we get one bit of $w$ for free
  - Negation is so fast that we can do it on the fly (saves half the table, faster constant-time lookups)
Using efficient negation

- So far everything we did works for any cyclic group $\langle P \rangle$
- Elliptic curves have so much more to offer
- For example, efficient negation: $-(x, y) = (x, -y)$ (on Weierstrass curves)
- Idea: use a signed representation for the scalar
- Fixed-window scalar multiplication:
  - Write scalar as $k_0, \ldots, k_{m-1}$ with $k_i \in [-2^w, \ldots, 2^w - 1]$
  - Precompute $T = (-2^w P, (-2^w + 1)P, \ldots, \mathcal{O}, P, \ldots, (2^w - 1)P$
  - Perform normal fixed-window scalar multiplication
  - Half of the precomputation is almost free, we get one bit of $w$ for free
  - Negation is so fast that we can do it on the fly (saves half the table, faster constant-time lookups)
- Similar scalar-negation speedup for sliding-window multiplication
Using other efficient endomorphisms

- Ben showed us before that there are efficient endomorphisms on elliptic curves
- Let’s now just take an efficient endomorphism \( \varphi \)
- Let’s assume that \( \varphi(Q) \) corresponds to \( \lambda Q \) for all \( Q \in \langle P \rangle \)

We can use this for faster scalar multiplication (Gallant, Lambert, Vanstone, 2000; and Galbraith, Lin, Scott, 2009)

Write scalar \( k = k_1 + k_2 \lambda \) with \( k_1 \) and \( k_2 \) half the length of \( k \)

Perform half-size double-scalar multiplication \( k_1(P) + k_2(\varphi(P)) \)

Save half of the doublings (estimated speedup: \( 30\% - 40\% \))

With two efficient endomorphisms we can do a 4-dimensional decomposition

Perform quarter-size quad-scalar multiplication (save another 25\% of doublings)
Using other efficient endomorphisms

- Ben showed us before that there are efficient endomorphisms on elliptic curves
- Let’s now just take an efficient endomorphism $\varphi$
- Let’s assume that $\varphi(Q)$ corresponds to $\lambda Q$ for all $Q \in \langle P \rangle$
- We can use this for faster scalar multiplication (Gallant, Lambert, Vanstone, 2000; and Galbraith, Lin, Scott, 2009)
  - Write scalar $k = k_1 + k_2\lambda$ with $k_1$ and $k_2$ half the length of $k$
  - Perform half-size double-scalar multiplication $k_1(P) + k_2(\varphi(P))$
  - Save half of the doublings (estimated speedup: $30 - 40\%$)
Ben showed us before that there are efficient endomorphisms on elliptic curves.

Let’s now just take an efficient endomorphism $\varphi$.

Let’s assume that $\varphi(Q)$ corresponds to $\lambda Q$ for all $Q \in \langle P \rangle$.

We can use this for faster scalar multiplication (Gallant, Lambert, Vanstone, 2000; and Galbraith, Lin, Scott, 2009)

- Write scalar $k = k_1 + k_2 \lambda$ with $k_1$ and $k_2$ half the length of $k$.
- Perform half-size double-scalar multiplication $k_1(P) + k_2(\varphi(P))$.
- Save half of the doublings (estimated speedup: $30 - 40\%$).

With two efficient endomorphisms we can do a 4-dimensional decomposition.

Perform quarter-size quad-scalar multiplication (save another $25\%$ of doublings).
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 \( P - Q \)
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 $R = P + Q$
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
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
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”

\[
\text{const } a_{24} = (A + 2)/4 \quad (A \text{ from the curve equation})
\]

\text{function } \text{ladderstep}(X_{Q-P}, X_P, Z_P, X_Q, Z_Q)

\[
\begin{align*}
t_1 & \leftarrow X_P + Z_P \\
t_6 & \leftarrow t_1^2 \\
t_2 & \leftarrow X_P - Z_P \\
t_7 & \leftarrow t_2^2 \\
t_5 & \leftarrow t_6 - t_7 \\
t_3 & \leftarrow X_Q + Z_Q \\
t_4 & \leftarrow X_Q - Z_Q \\
t_8 & \leftarrow t_4 \cdot t_1 \\
t_9 & \leftarrow t_3 \cdot t_2 \\
X_{P+Q} & \leftarrow (t_8 + t_9)^2 \\
Z_{P+Q} & \leftarrow X_{Q-P} \cdot (t_8 - t_9)^2 \\
X_{[2]P} & \leftarrow t_6 \cdot t_7 \\
Z_{[2]P} & \leftarrow t_5 \cdot (t_7 + a_{24} \cdot t_5)
\end{align*}
\]

\text{return } (X_{[2]P}, Z_{[2]P}, X_{P+Q}, Z_{P+Q})

\text{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: $(X[k]_P, Z[k]_P)$ fulfilling $x[k]_P = X[k]_P/Z[k]_P$

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

for $i \leftarrow n - 1$ downto 0 do
    if bit $i$ of $k$ is 1 then
        $(X_3, Z_3, X_2, Z_2) \leftarrow \text{ladderstep}(X_1, X_3, Z_3, X_2, Z_2)$
    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 $(X_2, Z_2)$
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)
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)
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
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 $n = 128$?
Multi-scalar multiplication

- Consider computation \( Q = \sum_{i=1}^{n} k_i P_i \)
- We looked at \( n = 2 \) before, how about \( n = 128? \)
- Idea: Assume \( k_1 > k_2 > \cdots > k_n \).
- Bos-Coster algorithm: recursively compute
  \[
  Q = (k_1 - k_2)P_1 + k_2(P_1 + P_2) + k_3P_3 \cdots + k_nP_n
  \]
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
- Each step “eliminates” expected $\log n$ scalar bits
- Can be very fast (but not constant-time)
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
- Each step “eliminates” expected $\log n$ scalar bits
- 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 $\lfloor (i - 1)/2 \rfloor$. 
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.
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.
Coming back to finite-field inversion

▶ Inversion with Fermat’s theorem uses exponentiation with $p - 2$
▶ Exponentiation is not really different from scalar multiplication (doublings become squarings, additions become multiplications)
Coming back to finite-field inversion

- Inversion with Fermat’s theorem uses exponentiation with $p - 2$
- Exponentiation is not really different from scalar multiplication (doublings become squarings, additions become multiplications)
- The prime $p$ is public, so also $p - 2$ is public
- First idea: use sliding window to compute exponentiation
Coming back to finite-field inversion

- Inversion with Fermat’s theorem uses exponentiation with \( p - 2 \)
- Exponentiation is not really different from scalar multiplication (doublings become squarings, additions become multiplications)
- The prime \( p \) is public, so also \( p - 2 \) is public
- First idea: use sliding window to compute exponentiation
- But wait, \( p \) is not only public, it’s a fixed system parameter, can we do better?
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 $s_i$ it holds that $s_i = s_j + s_k$ and $j, k < i$
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 $s_i$ it holds that $s_i = s_j + s_k$ and $j, k < i$

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

- Start with $s_1P = P$
- Compute $s_iP = s_jP + s_kP$ for $i = 2, \ldots, m$
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 $s_i$ it holds that $s_i = s_j + s_k$ and $j, k < i$

- An addition chain for $k$ immediately translates into a scalar multiplication algorithm to compute $kP$:
  - Start with $s_1P = P$
  - Compute $s_iP = s_jP + s_kP$ for $i = 2, \ldots, m$

- All algorithms so far basically just computed additions chains “on the fly”
- Signed-scalar representations are “addition-subtraction chains”
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 $s_i$ it holds that $s_i = s_j + s_k$ and $j, k < i$

- An addition chain for $k$ immediately translates into a scalar multiplication algorithm to compute $kP$:
  - Start with $s_1P = P$
  - Compute $s_iP = s_jP + s_kP$ for $i = 2, \ldots, m$

- All algorithms so far basically just computed additions chains “on the fly”
- Signed-scalar representations are “addition-subtraction chains”
- For inversion we know $k$ at compile time, we can spend a lot of time to find a good addition chain.
Inversion in $\mathbb{F}_{2^{255}-19}$

```c
void fe25519_invert(fe25519 *r, const fe25519 *x)
{
    fe25519 z2, z9, z11, z2_5_0, z2_10_0, z2_20_0, z2_50_0, z2_100_0, t;
    int i;
    /* 2 */ fe25519_square(&z2,x);
    /* 4 */ fe25519_square(&t,&z2);
    /* 8 */ fe25519_square(&t,&t);
    /* 9 */ fe25519_mul(&z9,&t,x);
    /* 11 */ fe25519_mul(&z11,&z9,&z2);
    /* 22 */ fe25519_square(&t,&z11);
    /* $2^5 - 2^0 = 31$ */fe25519_mul(&z2_5_0,&t,&z9);
    /* $2^6 - 2^1$ */ fe25519_square(&t,&z2_5_0);
    /* $2^{10} - 2^0$ */ fe25519_mul(&z2_10_0,&t,&z2_5_0);
    /* $2^{11} - 2^1$ */ fe25519_square(&t,&z2_10_0);
    /* $2^{20} - 2^{10}$ */ for (i = 1;i < 5;i++) { fe25519_square(&t,&t); }
    /* $2^20 - 2^0$ */ fe25519_mul(&z2_20_0,&t,&z2_10_0);
    /* $2^21 - 2^1$ */ fe25519_square(&t,&z2_20_0);
    /* $2^40 - 2^{20}$ */ for (i = 1;i < 20;i++) { fe25519_square(&t,&t); }
    /* $2^40 - 2^0$ */ fe25519_mul(&t,&t,&z2_20_0);
}
```

Scalar-multiplication algorithms 36
Inversion in $\mathbb{F}_{2^{255}-19}$

```c
/* 2^41 - 2^1 */ fe25519_square(&t,&t);
/* 2^50 - 2^10 */ for (i = 1; i < 10; i++) { fe25519_square(&t,&t); }
/* 2^50 - 2^0 */ fe25519_mul(&z2_50_0,&t,&z2_10_0);
/* 2^51 - 2^1 */ fe25519_square(&t,&z2_50_0);
/* 2^100 - 2^50 */ for (i = 1; i < 50; i++) { fe25519_square(&t,&t); }
/* 2^100 - 2^0 */ fe25519_mul(&z2_100_0,&t,&z2_50_0);
/* 2^101 - 2^1 */ fe25519_square(&t,&z2_100_0);
/* 2^200 - 2^100 */ for (i = 1; i < 100; i++) { fe25519_square(&t,&t); }
/* 2^200 - 2^0 */ fe25519_mul(&t,&t,&z2_100_0);
/* 2^201 - 2^1 */ fe25519_square(&t,&t);
/* 2^250 - 2^50 */ for (i = 1; i < 50; i++) { fe25519_square(&t,&t); }
/* 2^250 - 2^0 */ fe25519_mul(&t,&t,&z2_50_0);
/* 2^251 - 2^1 */ fe25519_square(&t,&t);
/* 2^252 - 2^2 */ fe25519_square(&t,&t);
/* 2^253 - 2^3 */ fe25519_square(&t,&t);
/* 2^254 - 2^4 */ fe25519_square(&t,&t);
/* 2^255 - 2^5 */ fe25519_square(&t,&t);
/* 2^255 - 21 */ fe25519_mul(r,&t,&z11);
}
```
Summary

- Remember double-and-add
- Remember not to use it (at least never with a secret scalar)
Summary

- Remember double-and-add
- Remember not to use it (at least never with a secret scalar)
- Keep in mind that writing constant-time code is hard
Summary

- Remember double-and-add
- Remember not to use it (at least never with a secret scalar)
- Keep in mind that writing constant-time code is hard
- A beer of your choice for anybody who computes $a^{2^{255}-21}$ in 254 squarings and 10 multiplications
Summary

- Remember double-and-add
- Remember not to use it (at least never with a secret scalar)
- Keep in mind that writing constant-time code is hard
- A beer of your choice for anybody who computes $a^{2^{255} - 21}$ in 254 squarings and 10 multiplications
- Two beers of your choice for anybody who computes $a^{2^{255} - 21}$ in 254 squarings and 9 multiplications
Summary

- Remember double-and-add
- Remember not to use it (at least never with a secret scalar)
- Keep in mind that writing constant-time code is hard
- A beer of your choice for anybody who computes $a^{2^{255} - 21}$ in $254$ squarings and $10$ multiplications
- Two beers of your choice for anybody who computes $a^{2^{255} - 21}$ in $254$ squarings and $9$ multiplications
- . . .
Summary

- Remember double-and-add
- Remember not to use it (at least never with a secret scalar)
- Keep in mind that writing constant-time code is hard
- A beer of your choice for anybody who computes $a^{2^{255} - 21}$ in 254 squarings and 10 multiplications
- Two beers of your choice for anybody who computes $a^{2^{255} - 21}$ in 254 squarings and 9 multiplications
- ... 
- Slides of both talks will be online at http://cryptojedi.org/