Software implementations of ECC: security and efficiency

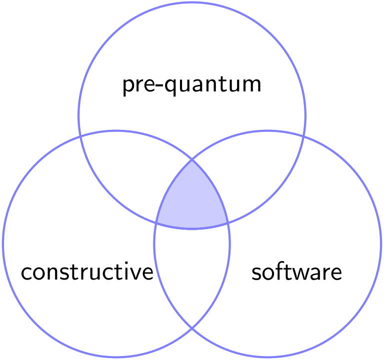
Tung Chou

Osaka University

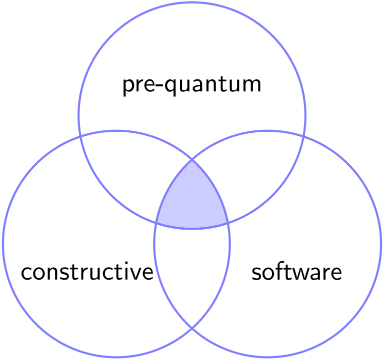
17 November 2018

Summer School of the ECC workshop

Scope



Scope



- : small exercises, homework

Diffie-Hellman key-exchange (agreement)

Parties agreed on: **group** $G = \langle g \rangle$

(e.g., $G \subset \mathbb{Z}_p^*$)

Diffie-Hellman key-exchange (agreement)

Parties agreed on: **group** $G = \langle g \rangle$

(e.g., $G \subset \mathbb{Z}_p^*$)

random a

$$\xrightarrow{\quad g^a \quad}$$

random b

$$\xleftarrow{\quad g^b \quad}$$

$s = (g^b)^a$ $\qquad$ $s = (g^a)^b$

Diffie-Hellman key-exchange (agreement)

Parties agreed on: **group** $G = \langle g \rangle$

(e.g., $G \subset \mathbb{Z}_p^*$)

random a

$$\xrightarrow{\quad g^a \quad}$$

random b

$$\xleftarrow{\quad g^b \quad}$$

$s = (g^b)^a$ $\qquad$ $s = (g^a)^b$

- "Assumed" to be "hard"
  - recovering c from g^c
  - recovering g^{ab} given g, g^a, g^b

Diffie-Hellman key-exchange (agreement)

Parties agreed on: **group** $G = \langle g \rangle$

(e.g., $G \subset \mathbb{Z}_p^*$)

random a

$\xrightarrow{\quad g^a \quad}$

random b

$\xleftarrow{\quad g^b \quad}$

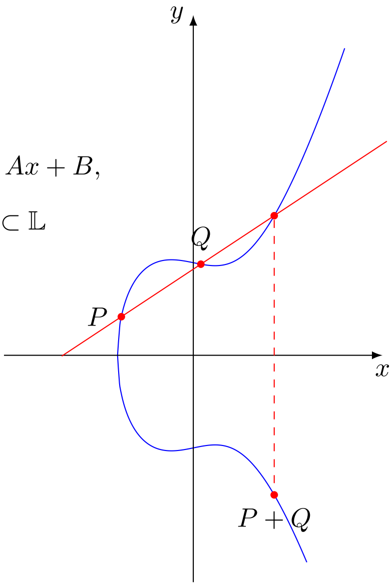
$s = (g^b)^a$ $\qquad$ $s = (g^a)^b$

- "Assumed" to be "hard"
  - recovering c from g^c
  - recovering g^{ab} given g, g^a, g^b
- Security depends on G

Elliptic curves

$$E: \; y^2 = x^3 + Ax + B,$$

$$A, B \in \mathbb{K} \subset \mathbb{L}$$



$$E(\mathbb{L}) = \{\infty\} \cup \{(x, y) \in \mathbb{L}^2 \mid y^2 = x^3 + Ax + B\}$$

Elliptic-curve Diffie-Hellman

Parties agreed on: **curve** E, $\langle B \rangle \subseteq E(\mathbb{K})$

random a

$$\xrightarrow{\quad aB \quad}$$

random b

$$\xleftarrow{\quad bB \quad}$$

$s = a(bB)$ $\qquad\qquad$ $s = b(aB)$

Elliptic-curve Diffie-Hellman

Parties agreed on: **curve** E, $\langle B \rangle \subseteq E(\mathbb{K})$

random a

$\xrightarrow{\quad aB \quad}$

random b

$\xleftarrow{\quad bB \quad}$

$s = a(bB)$ $s = b(aB)$

Elliptic-curve Diffie-Hellman

Parties agreed on: **curve** E, $\langle B \rangle \subseteq E(\mathbb{K})$

random a

$$\xrightarrow{\quad aB \quad}$$

random b

$$\xleftarrow{\quad bB \quad}$$

$s = a(bB)$ $\qquad$ $s = b(aB)$

"Assumed" to be "hard"

- recovering k from kP
- recovering abP given P, aP, bP

Scalar multiplications

$$k \cdot P$$

Scalar multiplications

$$k \cdot P$$

scalar

(secret? public?)

base point

(fixed? variable?)

safecurves.cr.yp.to/

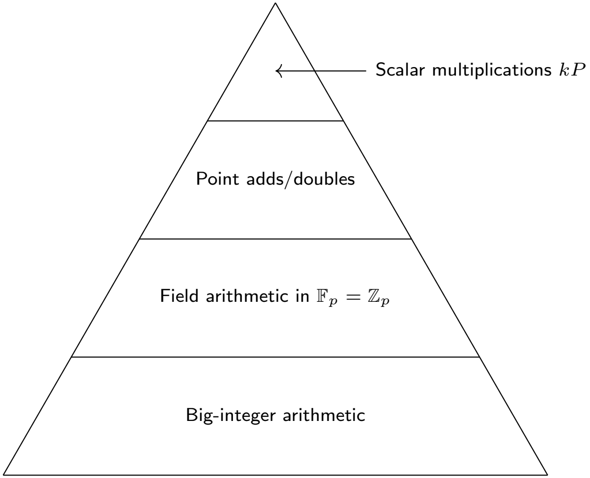
Curve	Safe?	Details
Anomalous	**False**	y^2 = x^3+15347898055371580590890576721314318823207531963035637503096292x+7444386449934505970367865204569124728350661870959593404279615 modulo p = 17676318486848893030961583018778670610489016512983351739677143 Created as an illustration of additive transfer and small discriminant.
M-221	True ✔	y^2 = x^3+117050x^2+x modulo p = 2^221 - 3 2013 Aranha–Barreto–Pereira–Ricardini (formerly named Curve2213)
E-222	True ✔	x^2+y^2 = 1+160102x^2y^2 modulo p = 2^222 - 117 2013 Aranha–Barreto–Pereira–Ricardini
NIST P-224	**False**	y^2 = x^3-3x+18958286285566608000408668544493926415504680968679321075787234672564 modulo p = 2^224 - 2^96 + 1 2000 NIST; also in SEC 2
Curve1174	True ✔	x^2+y^2 = 1-1174x^2y^2 modulo p = 2^251 - 9 2013 Bernstein–Hamburg–Krasnova–Lange
Curve25519	True ✔	y^2 = x^3+486662x^2+x modulo p = 2^255 - 19 2006 Bernstein
BN(2,254)	**False**	y^2 = x^3+0x+2 modulo p = 16798108731015832284940804142231733909889187121439069848933715426072753864723 2011 Pereira–Simplicio–Naehrig–Barreto pairing-friendly curve. Included as an illustration of multiplicative transfer and small discriminant.
brainpoolP256t1	**False**	y^2 = x^3-3x+46214326585032579593829631435610129746736367449296220983687490401182983727876 modulo p = 76884956397045344220809746629001649093037950200943055203735601445031516197751 2005 Brainpool
ANSSI FRP256v1	**False**	y^2 = x^3-3x+107744541122042688792155207242782455150382764043089114141096634497567301547839 modulo p = 109454571331697278617670725030735128145969349647868738157201323556196022393859 2011 ANSSI

safecurves.cr.yp.to/

Curve	Safe?	Details
Anomalous	False	y^2 = x^3+15347898055371580590890576721314318823207531963035637503096292x+7444386449934505970367865204569124728350661870959593404279615 modulo p = 17676318486848893030961583018778670610489016512983351739677143 Created as an illustration of additive transfer and small discriminant.
M-221	True ✔	y^2 = x^3+117050x^2+x modulo p = 2^221 - 3 2013 Aranha–Barreto–Pereira–Ricardini (formerly named Curve2213)
E-222	True ✔	x^2+y^2 = 1+160102x^2y^2 modulo p = 2^222 - 117 2013 Aranha–Barreto–Pereira–Ricardini
NIST P-224	False	y^2 = x^3-3x+18958286285566608000408668544493926415504680968679321075787234672564 modulo p = 2^224 - 2^96 + 1 2000 NIST; also in SEC 2
Curve1174	True ✔	x^2+y^2 = 1-1174x^2y^2 modulo p = 2^251 - 9 2013 Bernstein–Hamburg–Krasnova–Lange
Curve25519	True ✔	y^2 = x^3+486662x^2+x modulo p = 2^255 - 19 2006 Bernstein
BN(2,254)	False	y^2 = x^3+0x+2 modulo p = 16798108731015832284940804142231733909889187121439069848933715426072753864723 2011 Pereira–Simplicio–Naehrig–Barreto pairing-friendly curve. Included as an illustration of multiplicative transfer and small discriminant.
brainpoolP256t1	False	y^2 = x^3-3x+46214326585032579593829631435610129746736367449296220983687490401182983727876 modulo p = 76884956397045344220809746629001649093037950200943055203735601445031516197751 2005 Brainpool
ANSSI FRP256v1	False	y^2 = x^3-3x+107744541122042688792155207242782455150382764043089114141096634497567301547839 modulo p = 109454571331697278617670725030735128145969349647868738157201323556196022393859 2011 ANSSI

- Different curves forms
- Large field sizes: typically $> 2^{200}$

ECC implementation pyramid



www.hyperelliptic.org/EFD/

Explicit-Formulas Database

Bibliography

Genus-1 curves over large-characteristic fields

Doubling-oriented Doche–Icart–Kohel curves: $y^2=x^3+a*x^2+16*a*x$
Tripling-oriented Doche–Icart–Kohel curves: $y^2=x^3+3*a*(x+1)^2$
Edwards curves: $x^2+y^2=c^2*(1+d*x^2*y^2)$
Hessian curves: $x^3+y^3+1=3*d*x*y$
Jacobi intersections: $s^2+c^2=1$, $a*s^2+d^2=1$
Jacobi quartics: $y^2=x^4+2*a*x^2+1$
Montgomery curves: $b*y^2=x^3+a*x^2+x$
Short Weierstrass curves: $y^2=x^3+a*x+b$
Twisted Edwards curves: $a*x^2+y^2=1+d*x^2*y^2$
Twisted Hessian curves: $a*x^3+y^3+1=d*x*y$

Ordinary genus-1 curves over binary fields

Binary Edwards curves: $d1*(x+y)+d2*(x^2+y^2)=(x+x^2)*(y+y^2)$
Hessian curves: $x^3+y^3+1=3*d*x*y$
Short Weierstrass curves: $y^2+x*y=x^3+a2*x^2+a6$

www.hyperelliptic.org/EFD/

Explicit formulas for addition

The "mmadd-2008-hwcd-4" addition formulas [database entry; Sage verification script; Sage output; three-operand code]:

- Assumptions: Z1=1 and Z2=1.
- Cost: 6M + 8add + 2*2.
- Cost: 6M + 6add + 1*2 dependent upon the first point.
- Source: 2008 Hisil–Wong–Carter–Dawson, http://eprint.iacr.org/2008/522, Section 3.2, plus assumption Z1=1.
- Explicit formulas:

```
A = (Y1-X1)*(Y2+X2)
B = (Y1+X1)*(Y2-X2)
C = 2*T2
D = 2*T1
E = D+C
F = B-A
G = B+A
H = D-C
X3 = E*F
Y3 = G*H
T3 = E*H
Z3 = F*G
```

Naive scalar multiplication

- Input: scalar k, point P
- Output: kP

Naive scalar multiplication

- Input: scalar k, point P
- Output: kP

```
Q = P
For i from 1 to k-1 do
  Q += P
```

Naive scalar multiplication

- Input: scalar k, point P
- Output: kP

```
Q = P
For i from 1 to k-1 do
  Q += P
```

- Way too slow... (recall the group order)

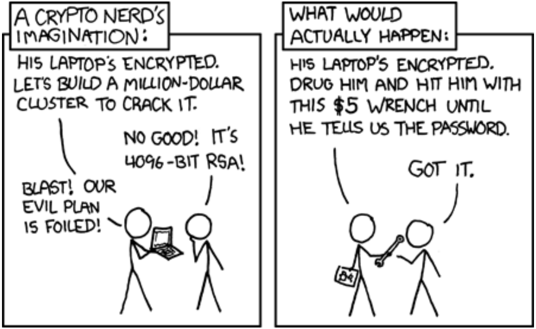
Naive scalar multiplication

- Input: scalar k, point P
- Output: kP

```
Q = P
For i from 1 to k-1 do
  Q += P
```

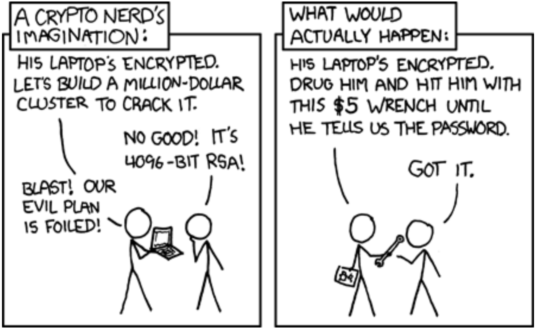
- Way too slow... (recall the group order)
- Timing depends on k (secret)!

Theory versus reality



(picture from xkcd.com/538/ [CC BY-NC 2.5])

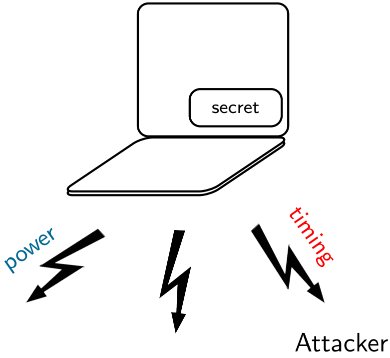
Theory versus reality



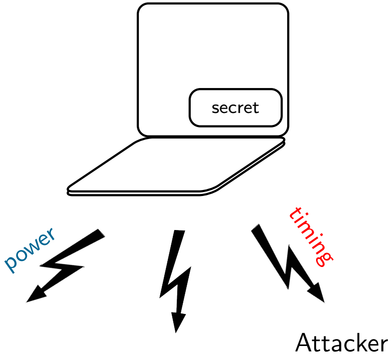
(picture from xkcd.com/538/ [CC BY-NC 2.5])

- How about the attack scenarios between the 2 cases?

Side channels

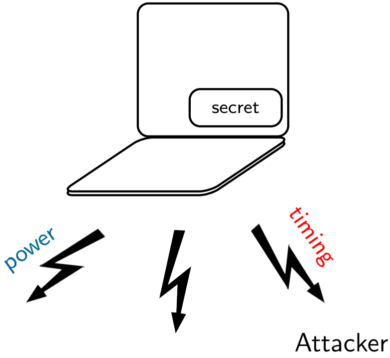


Side channels



- Side channel information might be correlated to the secret(s)

Side channels



- Side channel information might be correlated to the secret(s)
- We need **constant-time** implementations
  - timing should be independent of the secret(s)

Scalar multiplications with double-and-add

- Input: $k = (k_{\ell-1}k_{l-2}...k_0)_2$, point P
- Output: kP

Scalar multiplications with double-and-add

- Input: $k = (k_{\ell-1}k_{l-2}...k_0)_2$, point P
- Output: kP

```
Q = 0
For i from l-1 downto 0 do
  Q = 2Q
  if k[i] == 1
    Q += P
```

Scalar multiplications with double-and-add

- Input: $k = (k_{\ell-1}k_{l-2}...k_0)_2$, point P
- Output: kP

```
Q = 0
For i from l-1 downto 0 do
  Q = 2Q
  if k[i] == 1
    Q += P
```

- $\theta(\ell)$ instead of $k-1$ point operations
- Similarly, we can do “square-and-multiply”
- From LSB to MSB?

Scalar multiplications with double-and-add

- Input: $k = (k_{\ell-1}k_{l-2}...k_0)_2$, point P
- Output: kP

```
Q = 0
For i from l-1 downto 0 do
  Q = 2Q
  if k[i] == 1 ← timing difference!
    Q += P
```

- $\theta(\ell)$ instead of $k-1$ point operations
- Similarly, we can do "square-and-multiply"
- From LSB to MSB?

Scalar multiplications with double-and-add

- Input: $k = (k_{\ell-1}k_{\ell-2}...k_0)_2$, point P
- Output: kP

Scalar multiplications with double-and-add

- Input: $k = (k_{\ell-1}k_{\ell-2}...k_0)_2$, point P
- Output: kP

```
Q = 0
For i from l-1 downto 0 do
  R0 = 2Q
  R1 = Q + P
  if k[i] == 1
    Q = R1
  else
    Q = R0
```

Scalar multiplications with double-and-add

- Input: $k = (k_{\ell-1}k_{\ell-2}...k_0)_2$, point P
- Output: kP

```
Q = 0
For i from l-1 downto 0 do
  R0 = 2Q
  R1 = Q + P
  if k[i] == 1 ← branch prediction
    Q = R1
  else
    Q = R0
```

Scalar multiplications with double-and-add

- Input: $k = (k_{\ell-1}k_{\ell-2}...k_0)_2$, point P
- Output: kP

```
Q = 0
For i from l-1 downto 0 do
  R0 = 2Q
  R1 = Q + P
  if k[i] == 1 ← branch prediction
    Q = R1
  else
    Q = R0
```

(...and also the problem with instruction cache)

Branch prediction

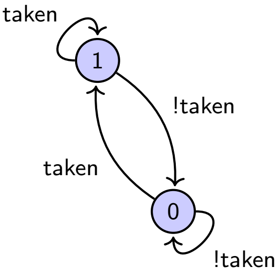
(1) : predict taken

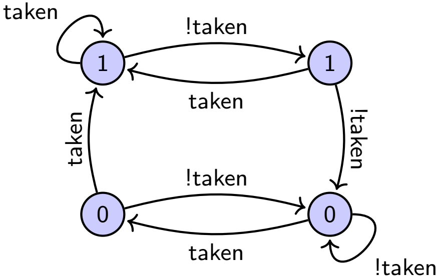
(0) : predict NOT taken

Branch prediction

1 : predict taken

0 : predict NOT taken

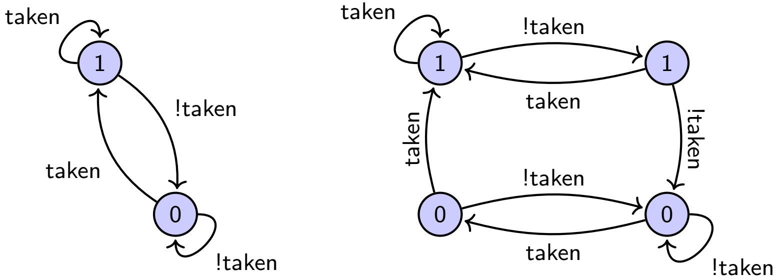




Branch prediction

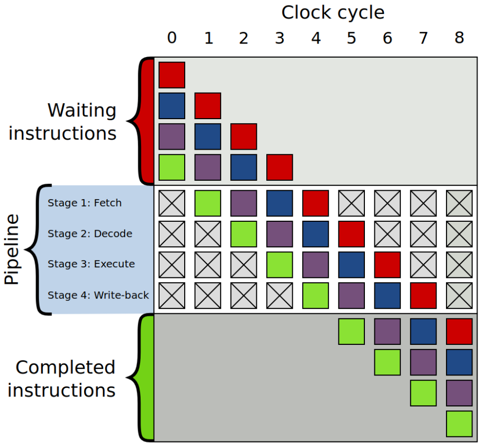
1 : predict taken

0 : predict NOT taken



- Good for loop structures. Why?

Branch prediction (cont.)



(picture from en.wikipedia.org/wiki/Branch_predictor [CC BY-SA 3.0])

Scalar multiplications with double-and-add

- Input: $k = (k_{\ell-1}k_{\ell-2}...k_0)_2$, point P
- Output: kP

```
Q = 0
For i from l-1 downto 0 do
  Q = 2Q
  R = Q + P

  Q = SEL(R, Q, k[i])
```
← constant-time

Scalar multiplications with double-and-add

- Input: $k = (k_{\ell-1}k_{\ell-2}...k_0)_2$, point P
- Output: kP

```
Q = 0
For i from l-1 downto 0 do
  Q = 2Q
  R = Q + P

  Q = SEL(R, Q, k[i])
```

$\leftarrow$ constant-time

```
SEL(R, Q, b):

  mask_R = -b
  mask_Q = ~mask_R

  return (mask_R & R) | (mask_Q & Q)
```

Scalar multiplications with double-and-add

- Input: $k = (k_{\ell-1}k_{\ell-2}...k_0)_2$, point P
- Output: kP

```
Q = 0
For i from l-1 downto 0 do
  Q = 2Q
  R = Q + P

  Q = SEL(R, Q, k[i])
```
$\leftarrow$ constant-time

```
SEL(R, Q, b):

  mask_R = -b
  mask_Q = ~mask_R

  return (mask_R & R) | (mask_Q & Q)
```

- Still many dummy operations

Scalar multiplications with table lookups (fixed window)

- Group w bits. Keep a table of $0P, \ldots, (2^w - 1)P$

Scalar multiplications with table lookups (fixed window)

- Group w bits. Keep a table of $0P, \ldots, (2^w - 1)P$
- Window width $w = 4$, $\ell = 256$.

Scalar multiplications with table lookups (fixed window)

- Group w bits. Keep a table of $0P, \ldots, (2^w - 1)P$
- Window width $w = 4$, $\ell = 256$.

```
For i from 0 to 15 do
  T[i] = iP

Q = 0
For i in [252, 248, ..., 0] do

  j = (k[i+3] k[i+2] k[i+1] k[i])_2

  Q = 2Q
  Q = 2Q
  Q = 2Q
  Q = 2Q

  Q += T[j]
```

Scalar multiplications with table lookups (fixed window)

- Group w bits. Keep a table of $0P, \ldots, (2^w - 1)P$
- Window width $w = 4$, $\ell = 256$.

```
For i from 0 to 15 do
  T[i] = iP

Q = 0
For i in [252, 248, ..., 0] do

  j = (k[i+3] k[i+2] k[i+1] k[i])_2

  Q = 2Q
  Q = 2Q
  Q = 2Q
  Q = 2Q

  Q += T[j]
```

- $\approx \ell/w$ additions

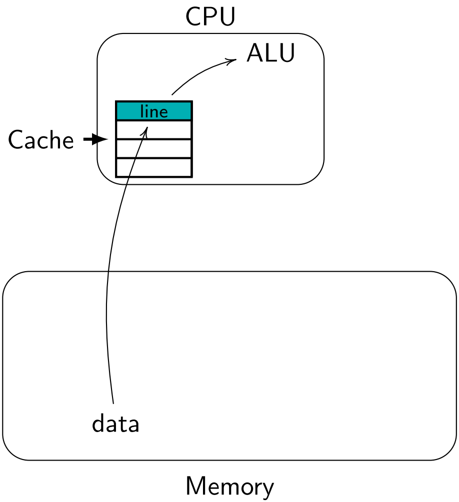
Scalar multiplications with table lookups (fixed window)

- Group w bits. Keep a table of $0P, \ldots, (2^w - 1)P$
- Window width $w = 4$, $\ell = 256$.

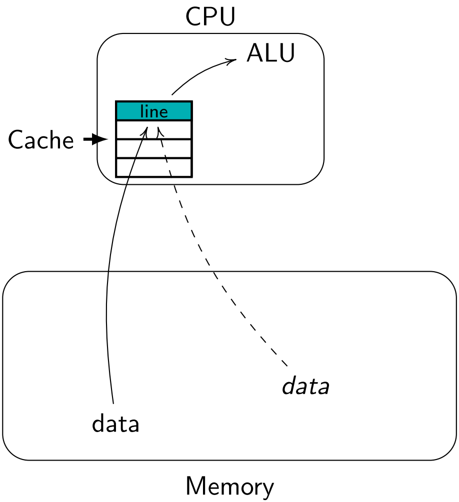
```
For i from 0 to 15 do
  T[i] = iP

Q = 0
For i in [252, 248, ..., 0] do

  j = (k[i+3] k[i+2] k[i+1] k[i])_2

  Q = 2Q
  Q = 2Q
  Q = 2Q
  Q = 2Q

  Q += T[j] ← non-constant-time!
```

- $\approx \ell/w$ additions

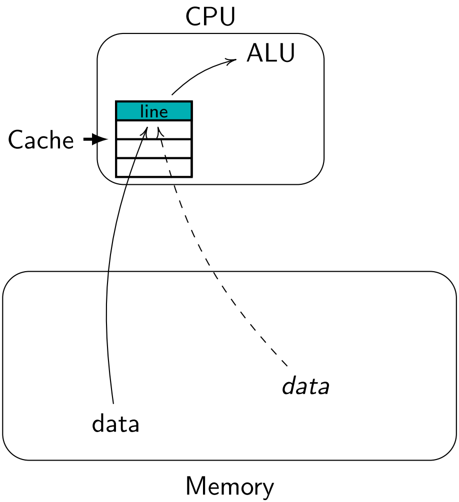
Caches



Caches

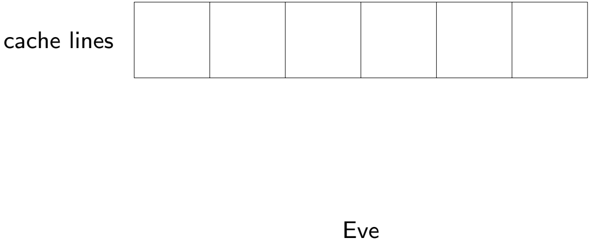


Caches

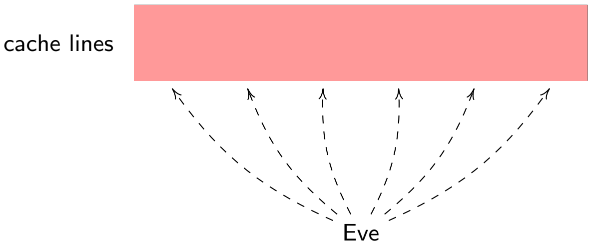


- Real architectures are more complex (hierarchy, associativity, etc)
- Instruction caches

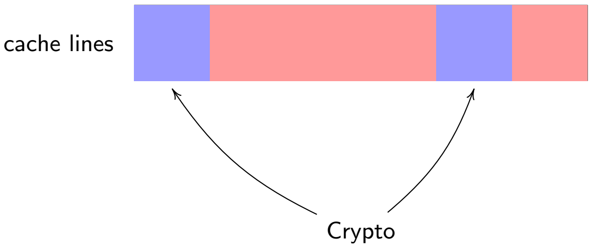
Cache-timing attacks



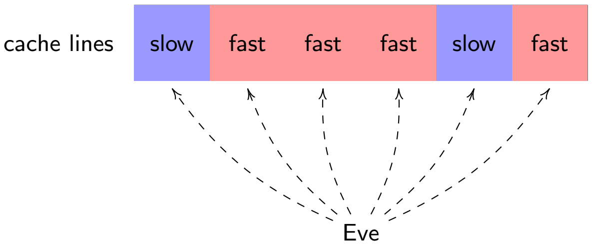
Cache-timing attacks



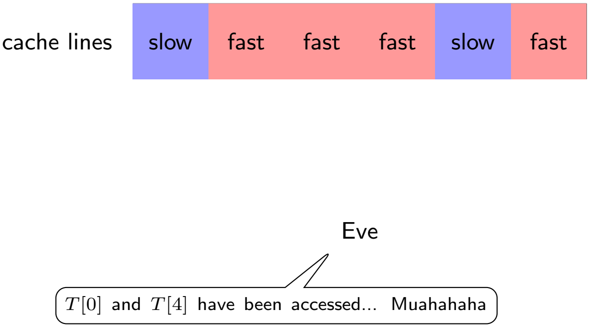
Cache-timing attacks



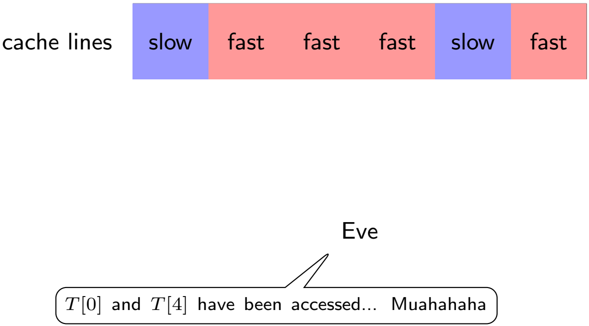
Cache-timing attacks



Cache-timing attacks



Cache-timing attacks



- Similar for instruction cache
- To avoid timing attacks, you should avoid
  - secret-dependent conditions
  - secret-dependent memory indices

Constant-time table lookups

- Consider $w = 3$

```
// compute mask_j from i s.t.
// mask_j = 0xFF..F if j == i
// mask_j = 0x00..0 if j != i

v  = mask_0 & T[0];
v |= mask_1 & T[1];
v |= mask_2 & T[2];
v |= mask_3 & T[3];
v |= mask_4 & T[4];
v |= mask_5 & T[5];
v |= mask_6 & T[6];
v |= mask_7 & T[7];

// now v = T[i]
```

Constant-time table lookups

- Consider $w = 3$

```
// compute mask_j from i s.t.
// mask_j = 0xFF..F if j == i
// mask_j = 0x00..0 if j != i

v  = mask_0 & T[0];
v |= mask_1 & T[1];
v |= mask_2 & T[2];
v |= mask_3 & T[3];
v |= mask_4 & T[4];
v |= mask_5 & T[5];
v |= mask_6 & T[6];
v |= mask_7 & T[7];

// now v = T[i]
```

- There is a limit on w

Signed window

- Each window is consider to be in $[-2^{w-1}, 2^{w-1}-1]$
- $\overline{1}$ denotes -1

$$(\underline{01}\ \underline{10}\ \underline{11}\ \underline{01}\ \underline{11})_2$$

Signed window

- Each window is consider to be in $[-2^{w-1}, 2^{w-1} - 1]$
- $\overline{1}$ denotes -1

$$(\underline{01}\ \underline{10}\ \underline{11}\ \underline{01}\ \underline{11})_2$$

$$\downarrow$$

$$(01\ 10\ 11\ 10\ 0\overline{1})_2$$

Signed window

- Each window is consider to be in $[-2^{w-1}, 2^{w-1} - 1]$
- $\overline{1}$ denotes -1

$$(\underline{01}\ \underline{10}\ \underline{11}\ \underline{01}\ \underline{11})_2$$

$$\downarrow$$

$$(01\ 10\ 11\ 10\ 0\overline{1})_2$$

$$\downarrow$$

$$(01\ 11\ 0\overline{1}\ 10\ 0\overline{1})_2$$

Signed window

- Each window is consider to be in $[-2^{w-1}, 2^{w-1}-1]$
- $\bar{1}$ denotes -1

$$(\underline{01}\ \underline{10}\ \underline{11}\ \underline{01}\ \underline{11})_2$$

$$\downarrow$$

$$(01\ 10\ 11\ 10\ 0\bar{1})_2$$

$$\downarrow$$

$$(01\ 11\ 0\bar{1}\ 10\ 0\bar{1})_2$$

$$\downarrow$$

$$(10\ 0\bar{1}\ 0\bar{1}\ 10\ 0\bar{1})_2$$

Signed window

- Each window is consider to be in $[-2^{w-1}, 2^{w-1}-1]$
- $\overline{1}$ denotes -1

$$(\underline{01}\ \underline{10}\ \underline{11}\ \underline{01}\ \underline{11})_2$$

$$\downarrow$$

$$(01\ 10\ 11\ {\color{blue}10\ 0\overline{1}})_2$$

$$\downarrow$$

$$(01\ {\color{blue}11\ 0\overline{1}}\ 10\ 0\overline{1})_2$$

$$\downarrow$$

$$({\color{blue}10\ 0\overline{1}}\ 0\overline{1}\ 10\ 0\overline{1})_2$$

- Rewriting is cheap

On-demand negation

```
// compute mask_j from i s.t.
// mask_j = 0xFF..F if |j| == i
// mask_j = 0x00..0 if |j| != i

v  = mask_0 & T[0];
v |= mask_1 & T[1];
v |= mask_2 & T[2];
v |= mask_3 & T[3];
v |= mask_4 & T[4];

v = SEL(-v, v, sign(j));

// now v = T[i]
```

- More memory-efficient
- More time-efficient (negation is cheap)

Fixed-base scalar multiplication

- Precomputation is free...
  - can use large window sizes (still many squarings).
  - can just precompute all $2^i P$ and do additions (no doublings).

Fixed-base scalar multiplication

- Precomputation is free...
  - can use large window sizes (still many squarings).
  - can just precompute all $2^i P$ and do additions (no doublings).

- Combining with windowing:
  - just precompute all $m2^{iw}P$ for $m \in [0, 2^w - 1]$.
  - $\approx \ell/w$ additions, no doublings

Fixed-base scalar multiplication

- Precomputation is free...
  - can use large window sizes (still many squarings).
  - can just precompute all $2^i P$ and do additions (no doublings).
- Combining with windowing:
  - just precompute all $m2^{iw}P$ for $m \in [0, 2^w - 1]$.
  - $\approx \ell/w$ additions, no doublings
- Suppose each table entry takes 64 bytes, $\ell = 256$ and $w = 4$. Memory requirement?

Variable-time instructions

					latency	
MULX	r32,r32,r32	3	3	p1 2p056	4	1
MULX	r32,r32,m32	3	4	p1 2p056 p23		1
MULX	r64,r64,r64	2	2	p1 p6	4	1
MULX	r64,r64,m64	2	3	p1 p6 p23		1
DIV	r8	9	9	p0 p1 p5 p6	22-25	9
DIV	r16	11	11	p0 p1 p5 p6	23-26	9
DIV	r32	10	10	p0 p1 p5 p6	22-29	9-11
DIV	r64	36	36	p0 p1 p5 p6	32-96	21-74
IDIV	r8	9	9	p0 p1 p5 p6	23-26	8
IDIV	r16	10	10	p0 p1 p5 p6	23-26	8
IDIV	r32	9	9	p0 p1 p5 p6	22-29	8-11
IDIV	r64	59	59	p0 p1 p5 p6	39-103	24-81

http://www.agner.org/optimize/instruction_tables.pdf

EdDSA signature scheme

B: based point, $\ell = |B|$, $\mathcal{H}$: hash function with $2b$-bit outputs

EdDSA signature scheme

B: based point, $\ell = |B|$, $\mathcal{H}$: hash function with $2b$-bit outputs

---

Key generation:

$$k: \text{ } b\text{-bit secret key}$$
$$a \leftarrow \mathcal{H}(k)$$
$$A \leftarrow aB: \text{ public key}$$

Signing:

$$r \leftarrow \mathcal{H}(a, M)$$
$$R \leftarrow rB$$
$$s \leftarrow (r + \mathcal{H}(R, A, M)a) \mod \ell$$
$$(R, s): \text{ signature of } M$$

Verification:

$$sB = R + \mathcal{H}(R, A, M)A$$

EdDSA signature scheme

B: based point, $\ell = |B|$, $\mathcal{H}$: hash function with $2b$-bit outputs

Key generation:

$$k: \text{ } b\text{-bit secret key}$$
$$a \leftarrow \mathcal{H}(k)$$
$$A \leftarrow aB: \text{ public key}$$

Signing:

$$r \leftarrow \mathcal{H}(a, M)$$
$$R \leftarrow rB$$
$$s \leftarrow (r + \mathcal{H}(R, A, M)a) \mod \ell$$
$$(R, s): \text{ signature of } M$$

Verification:

$$sB = R + \mathcal{H}(R, A, M)A$$

- Convince yourself that the scheme works.

Sliding window

- Keep mP with odd $m \in [0, 2^w - 1]$
- Shift the window if necessary

$$(\underline{01}\ \underline{10}\ \underline{11}\ \underline{01}\ \underline{11})_2$$

Sliding window

- Keep mP with odd $m \in [0, 2^w - 1]$
- Shift the window if necessary

$$(\underline{01}\ \underline{10}\ \underline{11}\ \underline{01}\ \underline{11})_2$$
$$\downarrow$$
$$(01\ 10\ 11\ 01\ 03)_2$$

Sliding window

- Keep mP with odd $m \in [0, 2^w - 1]$
- Shift the window if necessary

$$(\underline{01}\ \underline{10}\ \underline{11}\ \underline{01}\ \underline{11})_2$$

$$\downarrow$$

$$(01\ 10\ 11\ 01\ 03)_2$$

$$\downarrow$$

$$(01\ 10\ 11\ 01\ 03)_2$$

Sliding window

- Keep mP with odd $m \in [0, 2^w - 1]$
- Shift the window if necessary

$$(\underline{01}\ \underline{10}\ \underline{11}\ \underline{01}\ \underline{11})_2$$

$$\downarrow$$

$$(01\ 10\ 11\ 01\ 03)_2$$

$$\downarrow$$

$$(01\ 10\ 11\ 01\ 03)_2$$

$$\downarrow$$

$$(01\ 10\ 03\ 01\ 03)_2$$

Sliding window

- Keep mP with odd $m \in [0, 2^w - 1]$
- Shift the window if necessary

$$(\underline{01}\ \underline{10}\ \underline{11}\ \underline{01}\ \underline{11})_2$$
$$\downarrow$$
$$(01\ 10\ 11\ 01\ 03)_2$$
$$\downarrow$$
$$(01\ 10\ 11\ 01\ 03)_2$$
$$\downarrow$$
$$(01\ 10\ 03\ 01\ 03)_2$$
$$\downarrow$$
$$(00\ 30\ 03\ 01\ 03)_2$$

Sliding window

- Keep mP with odd $m \in [0, 2^w - 1]$
- Shift the window if necessary

$$(\underline{01}\ \underline{10}\ \underline{11}\ \underline{01}\ \underline{11})_2$$
$$\downarrow$$
$$(01\ 10\ 11\ 01\ 03)_2$$
$$\downarrow$$
$$(01\ 10\ 11\ 01\ 03)_2$$
$$\downarrow$$
$$(01\ 10\ 03\ 01\ 03)_2$$
$$\downarrow$$
$$(00\ 30\ 03\ 01\ 03)_2$$

- $4 \Rightarrow 3$ additions
- Can use signed sliding window

Double-scalar multiplications

- Input: public a, b, point P, Q
- Output: $aP + bQ$

Double-scalar multiplications

- Input: public a, b, point P, Q
- Output: $aP + bQ$

```
R = 0
For i from l-1 downto 0 do
  R = 2R
  if a[i] == 1
    R += P
  if b[i] == 1
    R += Q
```

Double-scalar multiplications

- Input: public a, b, point P, Q
- Output: $aP + bQ$

```
R = 0
For i from l-1 downto 0 do
  R = 2R
  if a[i] == 1
    R += P
  if b[i] == 1
    R += Q
```

- Sliding window can be applied (even different window sizes)
- Multi-scalar multiplication

Montgomery curves

1987 Montgomery:

- Curve

$$by^2 = x^3 + ax^2 + x$$

Montgomery curves

1987 Montgomery:

- Curve

$$by^2 = x^3 + ax^2 + x$$

- Doubling

$$2(x_2, y_2) = (x_4, y_4) \;\Rightarrow\; x_4 = \frac{(x_2^2 - 1)^2}{4x_2(x_2^2 + ax_2 + 1)}$$

Montgomery curves

1987 Montgomery:

- Curve

$$by^2 = x^3 + ax^2 + x$$

- Doubling

$$2(x_2, y_2) = (x_4, y_4) \;\Rightarrow\; x_4 = \frac{(x_2^2 - 1)^2}{4x_2(x_2^2 + ax_2 + 1)}$$

- **Differential** addition

$$(x_3, y_3) - (x_2, y_2) = (x_1, y_1)$$

$$(x_3, y_3) + (x_2, y_2) = (x_5, y_5)$$

$$\Rightarrow\; x_5 = \frac{(x_2 x_3 - 1)^2}{x_1(x_2 - x_3)^2}$$

Montgomery curves

1987 Montgomery:

- Curve

$$by^2 = x^3 + ax^2 + x$$

- Doubling

$$2(x_2, y_2) = (x_4, y_4) \;\Rightarrow\; x_4 = \frac{(x_2^2 - 1)^2}{4x_2(x_2^2 + ax_2 + 1)}$$

- **Differential** addition

$$(x_3, y_3) - (x_2, y_2) = (x_1, y_1)$$
$$(x_3, y_3) + (x_2, y_2) = (x_5, y_5)$$
$$\Rightarrow\; x_5 = \frac{(x_2x_3 - 1)^2}{x_1(x_2 - x_3)^2}$$

- We can compute the x coordinate of kP!

Montgomery curves

1987 Montgomery:

- Curve

$$by^2 = x^3 + ax^2 + x$$

- Doubling

$$2(x_2, y_2) = (x_4, y_4) \;\Rightarrow\; x_4 = \frac{(x_2^2 - 1)^2}{4x_2(x_2^2 + ax_2 + 1)}$$

- **Differential** addition

$$(x_3, y_3) - (x_2, y_2) = (x_1, y_1)$$

$$(x_3, y_3) + (x_2, y_2) = (x_5, y_5)$$

$$\Rightarrow\; x_5 = \frac{(x_2 x_3 - 1)^2}{x_1(x_2 - x_3)^2}$$

- We can compute the x coordinate of kP!

- Represent (x, y) as

$$(X : Z),\; x = X/Z$$

Montgomery ladder

Goal: compute $nP, n = (n_{\ell-1}n_{\ell-2}\cdots n_0)_2$

- Start from $(Q_\ell, R_\ell) = (0, P)$
- Maintain (Q_i, R_i) s.t. $R_i = Q_i + P$,

  $Q_i = \Sigma_{i=j}^{\ell} n_j 2^{j-i} P$ (i decreasing)
- Output $Q_0 = nP$

Montgomery ladder

Goal: compute $nP, n = (n_{\ell-1}n_{\ell-2}\cdots n_0)_2$

- Start from $(Q_\ell, R_\ell) = (0, P)$
- Maintain (Q_i, R_i) s.t. $R_i = Q_i + P$,

  $Q_i = \Sigma_{i=j}^{\ell} n_j 2^{j-i} P$ (i decreasing)
- Output $Q_0 = nP$

if $n_i = 0$,

$$(Q_i, R_i) = (2Q_{i+1}, Q_{i+1} + R_{i+1})$$

if $n_i = 1$,

$$(Q_i, R_i) = (Q_{i+1} + R_{i+1}, 2R_{i+1})$$

Montgomery ladder

Goal: compute $nP, n = (n_{\ell-1}n_{\ell-2}\cdots n_0)_2$

- Start from $(Q_\ell, R_\ell) = (0, P)$
- Maintain (Q_i, R_i) s.t. $R_i = Q_i + P$, $Q_i = \Sigma_{i=j}^{\ell} n_j 2^{j-i} P$ (i decreasing)
- Output $Q_0 = nP$

if $n_i = 0$,

$$(Q_i, R_i) = (2Q_{i+1}, Q_{i+1} + R_{i+1})$$

if $n_i = 1$,

$$(Q_i, R_i) = (Q_{i+1} + R_{i+1}, 2R_{i+1})$$

$n = (n_3n_2n_1n_0)_2 = (1101)_2 = 13$

Montgomery ladder

Goal: compute $nP, n = (n_{\ell-1}n_{\ell-2}\cdots n_0)_2$

- Start from $(Q_\ell, R_\ell) = (0, P)$
- Maintain (Q_i, R_i) s.t. $R_i = Q_i + P$,

  $Q_i = \Sigma_{i=j}^{\ell} n_j 2^{j-i} P$ (i decreasing)
- Output $Q_0 = nP$

if $n_i = 0$,

$$(Q_i, R_i) = (2Q_{i+1}, Q_{i+1}+R_{i+1})$$

if $n_i = 1$,

$$(Q_i, R_i) = (Q_{i+1}+R_{i+1}, 2R_{i+1})$$

$n = (n_3n_2n_1n_0)_2 = (1101)_2 = 13$

$(Q_4, \quad R_4) = (0, P)$

Montgomery ladder

Goal: compute $nP, n = (n_{\ell-1}n_{\ell-2}\cdots n_0)_2$

- Start from $(Q_\ell, R_\ell) = (0, P)$
- Maintain (Q_i, R_i) s.t. $R_i = Q_i + P$,

  $Q_i = \Sigma_{i=j}^{\ell} n_j 2^{j-i} P$ (i decreasing)
- Output $Q_0 = nP$

if $n_i = 0$,

$$(Q_i, R_i) = (2Q_{i+1}, Q_{i+1} + R_{i+1})$$

if $n_i = 1$,

$$(Q_i, R_i) = (Q_{i+1} + R_{i+1}, 2R_{i+1})$$

$n = (n_3 n_2 n_1 n_0)_2 = (1101)_2 = 13$

$(Q_4, \quad R_4) = (0, P)$

$n_3 = 1$

Montgomery ladder

Goal: compute $nP, n = (n_{\ell-1}n_{\ell-2}\cdots n_0)_2$

- Start from $(Q_\ell, R_\ell) = (0, P)$
- Maintain (Q_i, R_i) s.t. $R_i = Q_i + P$,

  $Q_i = \Sigma_{i=j}^{\ell} n_j 2^{j-i} P$ (i decreasing)
- Output $Q_0 = nP$

if $n_i = 0$,

$$(Q_i, R_i) = (2Q_{i+1}, Q_{i+1} + R_{i+1})$$

if $n_i = 1$,

$$(Q_i, R_i) = (Q_{i+1} + R_{i+1}, 2R_{i+1})$$

$n = (n_3 n_2 n_1 n_0)_2 = (1101)_2 = 13$

$(Q_4, \ R_4) = (0, P)$

$n_3 = 1$

$(Q_3, \ R_3) = (P, 2P)$

Montgomery ladder

Goal: compute $nP, n = (n_{\ell-1}n_{\ell-2}\cdots n_0)_2$

- Start from $(Q_\ell, R_\ell) = (0, P)$
- Maintain (Q_i, R_i) s.t. $R_i = Q_i + P$,

  $Q_i = \Sigma_{i=j}^{\ell} n_j 2^{j-i} P$ (i decreasing)
- Output $Q_0 = nP$

if $n_i = 0$,

$$(Q_i, R_i) = (2Q_{i+1}, Q_{i+1} + R_{i+1})$$

if $n_i = 1$,

$$(Q_i, R_i) = (Q_{i+1} + R_{i+1}, 2R_{i+1})$$

$n = (n_3 n_2 n_1 n_0)_2 = (1101)_2 = 13$

$$(Q_4, R_4) = (0, P)$$

$n_3 = 1$

$$(Q_3, R_3) = (P, 2P)$$

$n_2 = 1$

$$(Q_2, R_2) = (3P, 4P)$$

Montgomery ladder

Goal: compute $nP, n = (n_{\ell-1}n_{\ell-2}\cdots n_0)_2$

- Start from $(Q_\ell, R_\ell) = (0, P)$
- Maintain (Q_i, R_i) s.t. $R_i = Q_i + P$,

  $Q_i = \Sigma_{i=j}^{\ell} n_j 2^{j-i} P$ (i decreasing)
- Output $Q_0 = nP$

if $n_i = 0$,

$$(Q_i, R_i) = (2Q_{i+1}, Q_{i+1} + R_{i+1})$$

if $n_i = 1$,

$$(Q_i, R_i) = (Q_{i+1} + R_{i+1}, 2R_{i+1})$$

$n = (n_3 n_2 n_1 n_0)_2 = (1101)_2 = 13$

$(Q_4, \ R_4) = (0, P)$

$n_3 = 1$

$(Q_3, \ R_3) = (P, 2P)$

$n_2 = 1$

$(Q_2, \ R_2) = (3P, 4P)$

$n_1 = 0$

Montgomery ladder

Goal: compute $nP, n = (n_{\ell-1}n_{\ell-2}\cdots n_0)_2$

- Start from $(Q_\ell, R_\ell) = (0, P)$
- Maintain (Q_i, R_i) s.t. $R_i = Q_i + P$,

  $Q_i = \Sigma_{i=j}^{\ell} n_j 2^{j-i} P$ (i decreasing)
- Output $Q_0 = nP$

if $n_i = 0$,

$$(Q_i, R_i) = (2Q_{i+1}, Q_{i+1} + R_{i+1})$$

if $n_i = 1$,

$$(Q_i, R_i) = (Q_{i+1} + R_{i+1}, 2R_{i+1})$$

$n = (n_3 n_2 n_1 n_0)_2 = (1101)_2 = 13$

$$(Q_4, \ R_4) = (0, P)$$

$n_3 = 1$

$$(Q_3, \ R_3) = (P, 2P)$$

$n_2 = 1$

$$(Q_2, \ R_2) = (3P, 4P)$$

$n_1 = 0$

$$(Q_1, \ R_1) = (6P, 7P)$$

Montgomery ladder

Goal: compute $nP, n = (n_{\ell-1}n_{\ell-2}\cdots n_0)_2$

- Start from $(Q_\ell, R_\ell) = (0, P)$
- Maintain (Q_i, R_i) s.t. $R_i = Q_i + P$,

  $Q_i = \Sigma_{i=j}^{\ell} n_j 2^{j-i} P$ (i decreasing)
- Output $Q_0 = nP$

if $n_i = 0$,

$$(Q_i, R_i) = (2Q_{i+1}, Q_{i+1} + R_{i+1})$$

if $n_i = 1$,

$$(Q_i, R_i) = (Q_{i+1} + R_{i+1}, 2R_{i+1})$$

$n = (n_3n_2n_1n_0)_2 = (1101)_2 = 13$

$(Q_4, R_4) = (0, P)$

$n_3 = 1$

$(Q_3, R_3) = (P, 2P)$

$n_2 = 1$

$(Q_2, R_2) = (3P, 4P)$

$n_1 = 0$

$(Q_1, R_1) = (6P, 7P)$

$n_0 = 1$

Montgomery ladder

Goal: compute $nP, n = (n_{\ell-1} n_{\ell-2} \cdots n_0)_2$

- Start from $(Q_\ell, R_\ell) = (0, P)$
- Maintain (Q_i, R_i) s.t. $R_i = Q_i + P$,

  $Q_i = \Sigma_{i=j}^{\ell} n_j 2^{j-i} P$ (i decreasing)
- Output $Q_0 = nP$

if $n_i = 0$,

$$(Q_i, R_i) = (2Q_{i+1}, Q_{i+1} + R_{i+1})$$

if $n_i = 1$,

$$(Q_i, R_i) = (Q_{i+1} + R_{i+1}, 2R_{i+1})$$

$n = (n_3 n_2 n_1 n_0)_2 = (1101)_2 = 13$

$(Q_4, R_4) = (0, P)$

$n_3 = 1$

$(Q_3, R_3) = (P, 2P)$

$n_2 = 1$

$(Q_2, R_2) = (3P, 4P)$

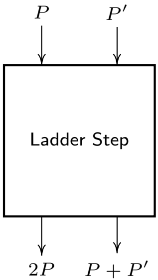
$n_1 = 0$

$(Q_1, R_1) = (6P, 7P)$

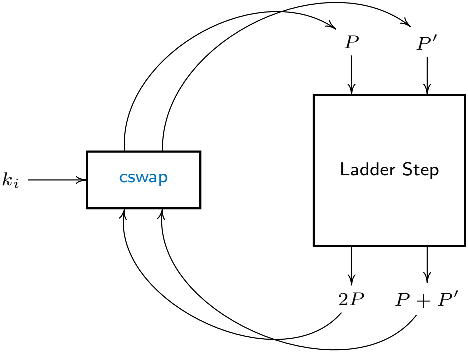
$n_0 = 1$

$(Q_0, R_0) = (13P, 14P)$

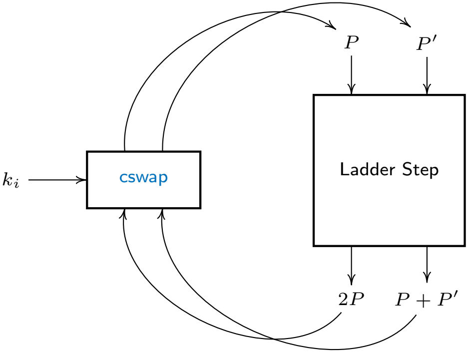
Implementing the Montgomery ladder



Implementing the Montgomery ladder



Implementing the Montgomery ladder



- How to implement cswap?
  - hint: 4 bit operations to conditionally swap 2 bits

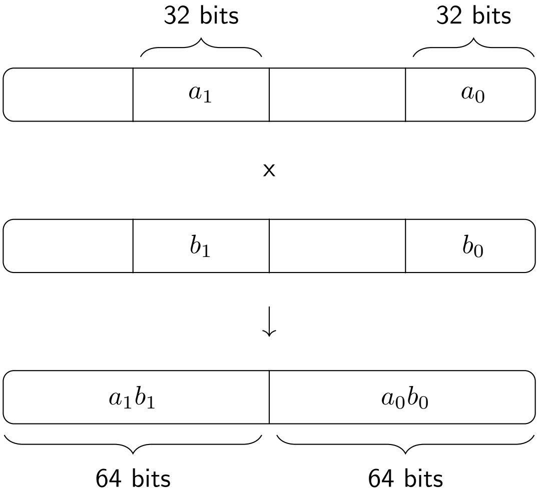
Vector units

- Advanced Vector Extensions (AVX)

511 256	255 128	127 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15

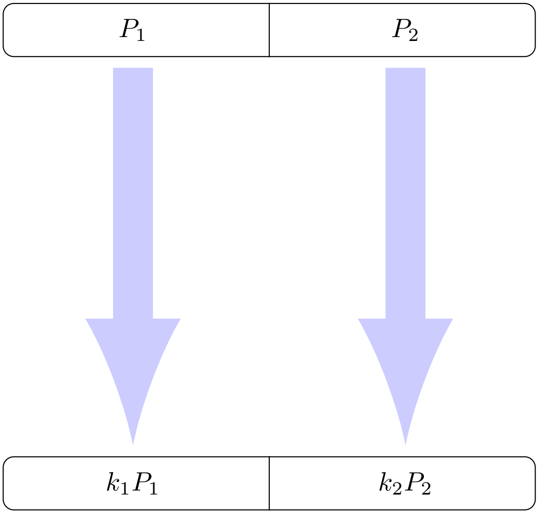
- Enables vector instructions

2-way vectorization

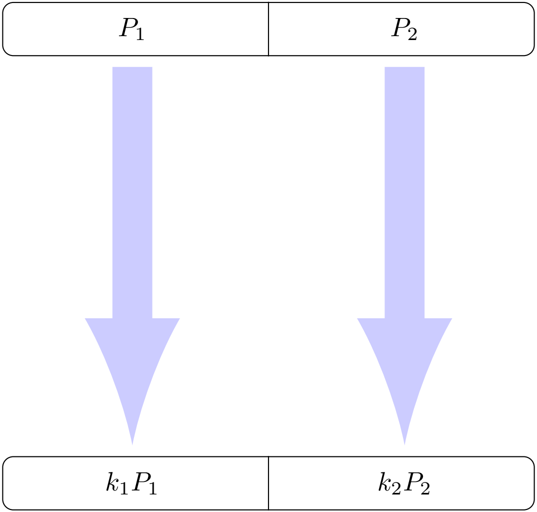


- Useful for accelerating cryptographic computation

Exploiting parallelism: naive way



Exploiting parallelism: naive way



- Might be good for busy servers. Other applications?

Exploiting parallelism: key generation

$$P = s_0 B + s_1 16 B + s_2 16^2 B + \cdots + s_{63} 16^{63} B$$

Exploiting parallelism: key generation

$$P = s_0 B + s_1 16 B + s_2 16^2 B + \cdots + s_{63} 16^{63} B$$

$$\downarrow$$

$$P_0 = s_0 B + s_2 16^2 B + \cdots + s_{62} 16^{62} B$$

$$P_1 = s_1 B + s_3 16^2 B + \cdots + s_{63} 16^{62} B$$

$$P = 16 P_1 + P_0$$

Exploiting parallelism: key generation

$$P = s_0 B + s_1 16 B + s_2 16^2 B + \cdots + s_{63} 16^{63} B$$

$$\downarrow$$

$$P_0 = s_0 B + s_2 16^2 B + \cdots + s_{62} 16^{62} B$$

$$P_1 = s_1 B + s_3 16^2 B + \cdots + s_{63} 16^{62} B$$

$$P = 16 P_1 + P_0$$

Exploiting parallelism: key generation

$$P = s_0 B + s_1 16B + s_2 16^2 B + \cdots + s_{63} 16^{63} B$$

$$\downarrow$$

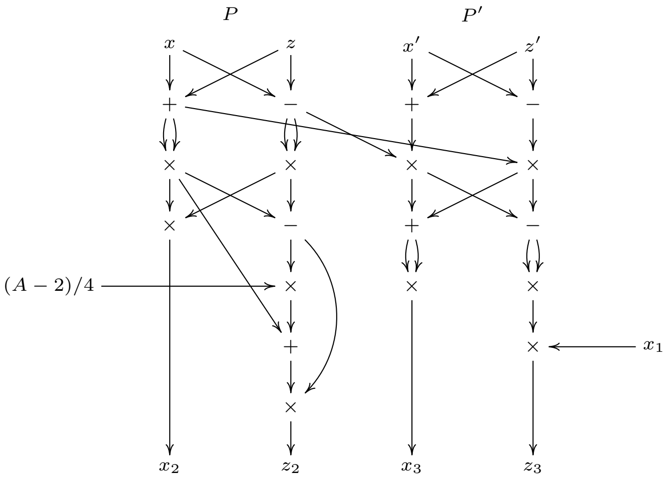
$$P_0 = s_0 B + s_2 16^2 B + \cdots + s_{62} 16^{62} B$$

$$P_1 = s_1 B + s_3 16^2 B + \cdots + s_{63} 16^{62} B$$

$$P = 16 P_1 + P_0$$

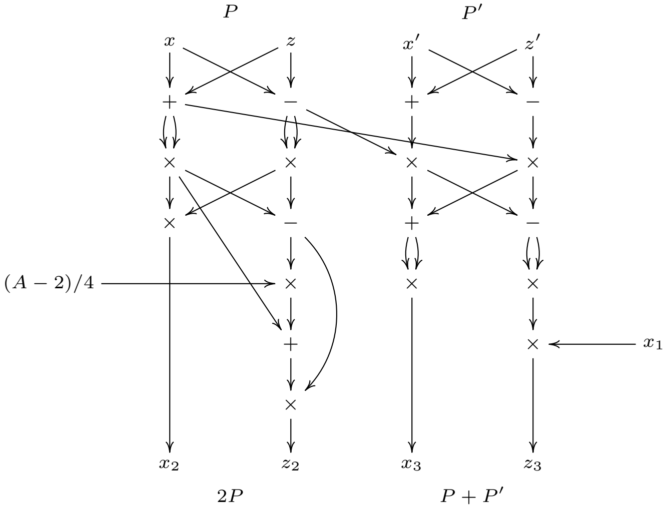
- Extra benefit of reducing table size
- Tradeoffs between time and space

Exploiting parallelism: ladder step



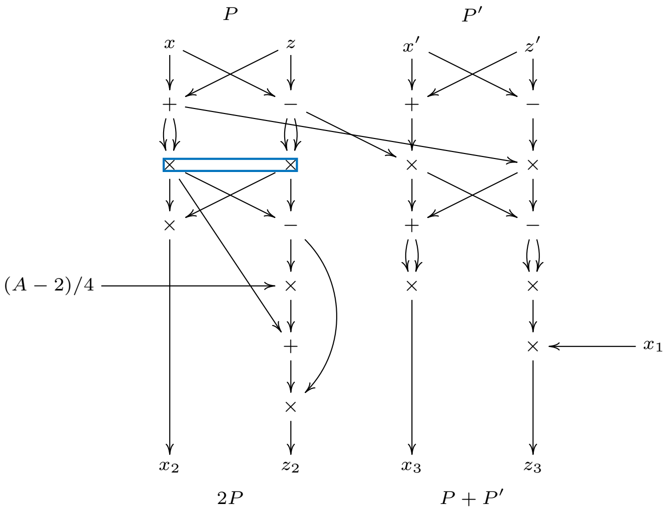
www.hyperelliptic.org/EFD/g1p/auto-montgom-xz.html#ladder-mladd-1987-m

Exploiting parallelism: ladder step



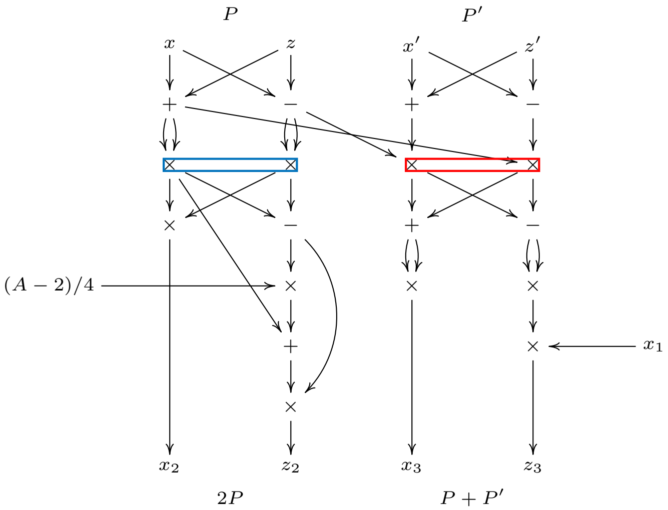
www.hyperelliptic.org/EFD/g1p/auto-montgom-xz.html#ladder-mladd-1987-m

Exploiting parallelism: ladder step



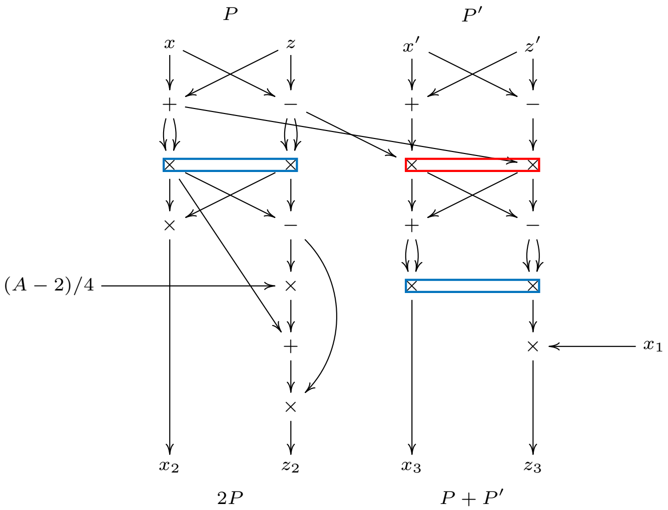
www.hyperelliptic.org/EFD/g1p/auto-montgom-xz.html#ladder-mladd-1987-m

Exploiting parallelism: ladder step



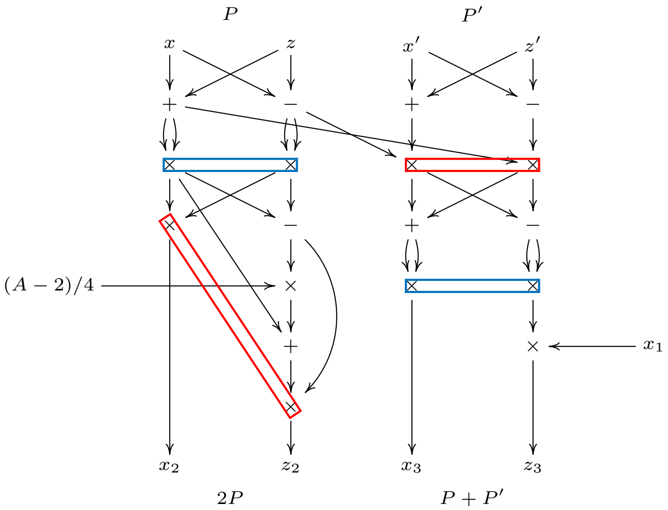
www.hyperelliptic.org/EFD/g1p/auto-montgom-xz.html#ladder-mladd-1987-m

Exploiting parallelism: ladder step



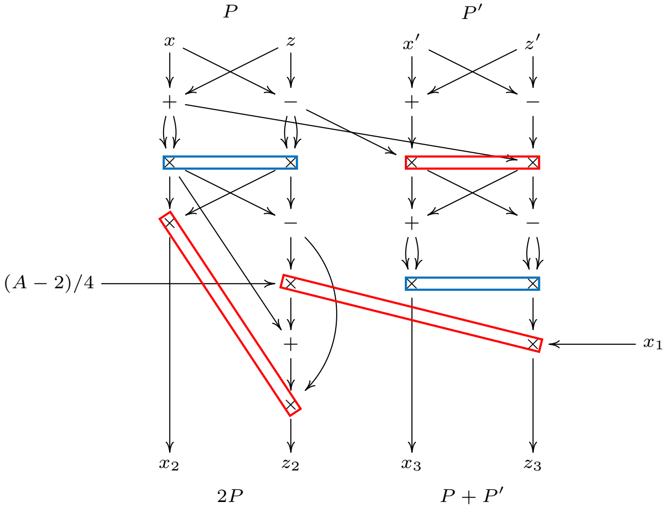
www.hyperelliptic.org/EFD/g1p/auto-montgom-xz.html#ladder-mladd-1987-m

Exploiting parallelism: ladder step



www.hyperelliptic.org/EFD/g1p/auto-montgom-xz.html#ladder-mladd-1987-m

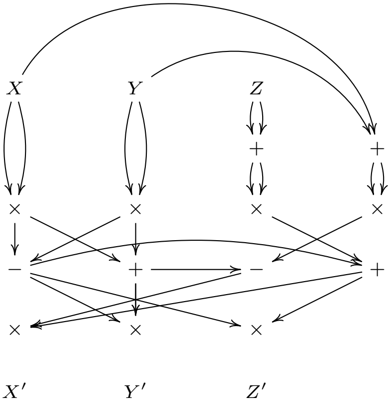
Exploiting parallelism: ladder step



www.hyperelliptic.org/EFD/g1p/auto-montgom-xz.html#ladder-mladd-1987-m

Exploiting parallelism: doubling (twisted Edwards curves)

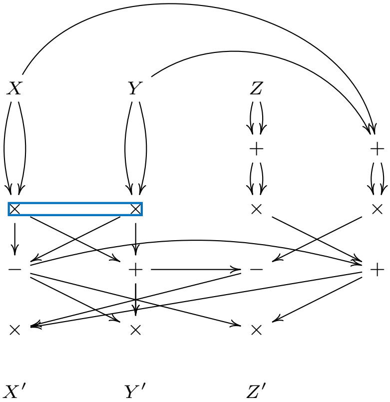
- Let $(X : Y : Z) \coloneqq (X/Z, Y/Z)$
- Goal: compute $(X' : Y' : Z') = 2(X : Y : Z)$



www.hyperelliptic.org/EFD/g1p/auto-twisted-extended-1.html#doubling-dbl-2008-hwcd

Exploiting parallelism: doubling (twisted Edwards curves)

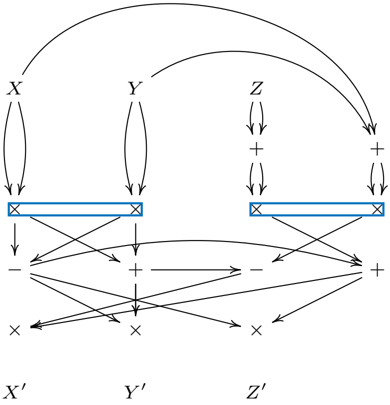
- Let $(X : Y : Z) \coloneqq (X/Z, Y/Z)$
- Goal: compute $(X' : Y' : Z') = 2(X : Y : Z)$



www.hyperelliptic.org/EFD/g1p/auto-twisted-extended-1.html#doubling-dbl-2008-hwcd

Exploiting parallelism: doubling (twisted Edwards curves)

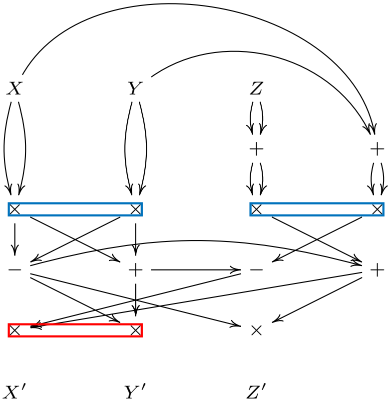
- Let $(X : Y : Z) \coloneqq (X/Z, Y/Z)$
- Goal: compute $(X' : Y' : Z') = 2(X : Y : Z)$



www.hyperelliptic.org/EFD/g1p/auto-twisted-extended-1.html#doubling-dbl-2008-hwcd

Exploiting parallelism: doubling (twisted Edwards curves)

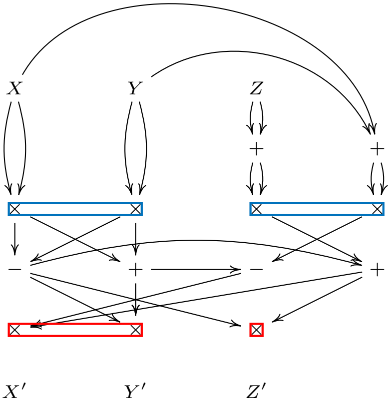
- Let $(X : Y : Z) \coloneqq (X/Z, Y/Z)$
- Goal: compute $(X' : Y' : Z') = 2(X : Y : Z)$



www.hyperelliptic.org/EFD/g1p/auto-twisted-extended-1.html#doubling-dbl-2008-hwcd

Exploiting parallelism: doubling (twisted Edwards curves)

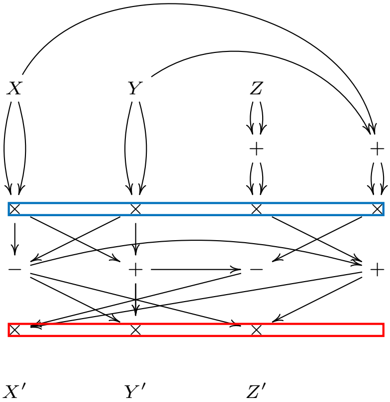
- Let $(X : Y : Z) \coloneqq (X/Z, Y/Z)$
- Goal: compute $(X' : Y' : Z') = 2(X : Y : Z)$



www.hyperelliptic.org/EFD/g1p/auto-twisted-extended-1.html#doubling-dbl-2008-hwcd

Exploiting parallelism: doubling (twisted Edwards curves)

- Let $(X : Y : Z) \coloneqq (X/Z, Y/Z)$
- Goal: compute $(X' : Y' : Z') = 2(X : Y : Z)$



www.hyperelliptic.org/EFD/g1p/auto-twisted-extended-1.html#doubling-dbl-2008-hwcd

Field representation: case of $\mathbb{F}_{2^{255}-19}$

- Intuition: radix-2^{64} representation
  - each field elements in represented by 4 **limbs**

$$f = \quad f_0 + \quad f_1 2^{64} + \quad f_2 2^{128} + \quad f_3 2^{192}$$

Field representation: case of $\mathbb{F}_{2^{255}-19}$

- Intuition: radix-2^{64} representation
  - each field elements in represented by 4 **limbs**

$$
\begin{array}{lllll}
f = & f_0 + & f_1 2^{64} + & f_2 2^{128} + & f_3 2^{192} \\
g = & g_0 + & g_1 2^{64} + & g_2 2^{128} + & g_3 2^{192}
\end{array}
$$

Field representation: case of $\mathbb{F}_{2^{255}-19}$

- Intuition: radix-2^{64} representation
  - each field elements in represented by 4 **limbs**

$$
\begin{aligned}
f = \quad & f_0 + \quad f_1 2^{64} + \quad f_2 2^{128} + \quad f_3 2^{192} \\
g = \quad & g_0 + \quad g_1 2^{64} + \quad g_2 2^{128} + \quad g_3 2^{192}
\end{aligned}
$$

- Multiplication $(\mathrm{mod}\ 2^{256} - 38)$:

$$
\begin{aligned}
h_0 &= f_0 g_0 + 38 f_1 g_3 + 38 f_2 g_2 + 38 f_3 g_1 \\
h_1 &= f_0 g_1 + f_1 g_0 + 38 f_2 g_3 + 38 f_3 g_2 \\
h_2 &= f_0 g_2 + f_1 g_1 + f_2 g_0 + 38 f_3 g_3 \\
h_3 &= f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0
\end{aligned}
$$

Field representation: case of $\mathbb{F}_{2^{255}-19}$

- Intuition: radix-2^{64} representation
  - each field elements in represented by 4 **limbs**

$$
\begin{array}{rrrrr}
f = & f_0 + & f_1 2^{64} + & f_2 2^{128} + & f_3 2^{192} \\
g = & g_0 + & g_1 2^{64} + & g_2 2^{128} + & g_3 2^{192}
\end{array}
$$

- Multiplication $\pmod{2^{256} - 38}$:

$$
\begin{array}{rrrrr}
h_0 = & f_0 g_0 + & 38 f_1 g_3 + & 38 f_2 g_2 + & 38 f_3 g_1 \\
h_1 = & f_0 g_1 + & f_1 g_0 + & 38 f_2 g_3 + & 38 f_3 g_2 \\
h_2 = & f_0 g_2 + & f_1 g_1 + & f_2 g_0 + & 38 f_3 g_3 \\
h_3 = & f_0 g_3 + & f_1 g_2 + & f_2 g_1 + & f_3 g_0
\end{array}
$$

- Making use `MULX`: “$64 \times 64 \rightarrow 64\ 64$”

Field representation: case of $\mathbb{F}_{2^{255}-19}$

- Intuition: radix-2^{64} representation
  - each field elements in represented by 4 **limbs**

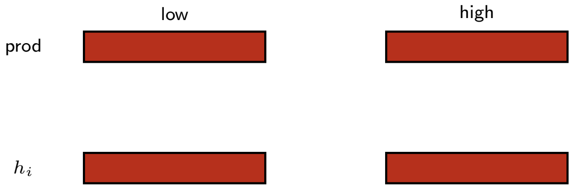
$$
\begin{aligned}
f &= f_0 + f_1 2^{64} + f_2 2^{128} + f_3 2^{192} \\
g &= g_0 + g_1 2^{64} + g_2 2^{128} + g_3 2^{192}
\end{aligned}
$$

- Multiplication $(\mathrm{mod}\ 2^{256} - 38)$:

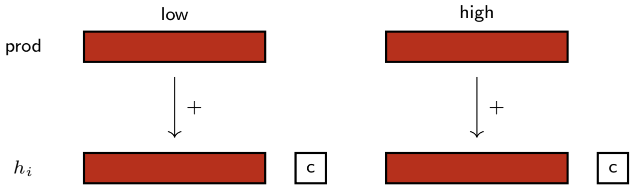
$$
\begin{aligned}
h_0 &= f_0g_0 + 38f_1g_3 + 38f_2g_2 + 38f_3g_1 \\
h_1 &= f_0g_1 + f_1g_0 + 38f_2g_3 + 38f_3g_2 \\
h_2 &= f_0g_2 + f_1g_1 + f_2g_0 + 38f_3g_3 \\
h_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0
\end{aligned}
$$

- Making use `MULX`: "$64 \times 64 \rightarrow 64\ 64$"
- Lots of carries!

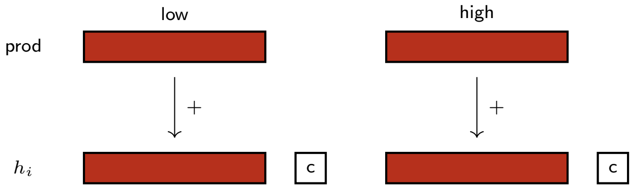
Adding products of the limbs



Adding products of the limbs



Adding products of the limbs



- adding 128-bit products produces 2 carry bits:

  $\rightarrow$ require 2 “addtion with carry” (`adc`) instructions
- `adc` can be expensive:

  6x slower than `add` in terms of througput on some Intel CPUs

Redundant representation

- Radix-51 representation

$$f = \quad f_0 + \quad f_1 2^{51} + \quad f_2 2^{102} + \quad f_3 2^{153} + \quad f_4 2^{204}$$

Redundant representation

- Radix-51 representation

$$
\begin{array}{llllll}
f = & f_0 + & f_1 2^{51} + & f_2 2^{102} + & f_3 2^{153} + & f_4 2^{204} \\
g = & g_0 + & g_1 2^{51} + & g_2 2^{102} + & g_3 2^{153} + & g_4 2^{204}
\end{array}
$$

Redundant representation

- Radix-51 representation

$$\begin{aligned} f &= f_0 + f_1 2^{51} + f_2 2^{102} + f_3 2^{153} + f_4 2^{204} \\ g &= g_0 + g_1 2^{51} + g_2 2^{102} + g_3 2^{153} + g_4 2^{204} \end{aligned}$$

- Multiplication

$$\begin{aligned} h_0 &= f_0 g_0 + 19 f_1 g_4 + 19 f_2 g_3 + 19 f_3 g_2 + 19 f_4 g_1 \\ h_1 &= f_0 g_1 + f_1 g_0 + 19 f_2 g_4 + 19 f_3 g_3 + 19 f_4 g_2 \\ h_2 &= f_0 g_2 + f_1 g_1 + f_2 g_0 + 19 f_3 g_4 + 19 f_4 g_3 \\ h_3 &= f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0 + 19 f_4 g_4 \\ h_4 &= f_0 g_4 + f_1 g_3 + f_2 g_2 + f_3 g_1 + f_4 g_0 \end{aligned}$$

Redundant representation

- Radix-51 representation

$$
\begin{aligned}
f = \quad & f_0 + \quad f_1 2^{51} + \quad f_2 2^{102} + \quad f_3 2^{153} + \quad f_4 2^{204} \\
g = \quad & g_0 + \quad g_1 2^{51} + \quad g_2 2^{102} + \quad g_3 2^{153} + \quad g_4 2^{204}
\end{aligned}
$$

- Multiplication

$$
\begin{aligned}
h_0 &= f_0 g_0 + 19 f_1 g_4 + 19 f_2 g_3 + 19 f_3 g_2 + 19 f_4 g_1 \\
h_1 &= f_0 g_1 + f_1 g_0 + 19 f_2 g_4 + 19 f_3 g_3 + 19 f_4 g_2 \\
h_2 &= f_0 g_2 + f_1 g_1 + f_2 g_0 + 19 f_3 g_4 + 19 f_4 g_3 \\
h_3 &= f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0 + 19 f_4 g_4 \\
h_4 &= f_0 g_4 + f_1 g_3 + f_2 g_2 + f_3 g_1 + f_4 g_0
\end{aligned}
$$

- Adding upper 64 bits does not generate carries!

  $\Rightarrow$ reducing (roughly) half of the `adc`'s

Reduction (carries)

- Carry $h_i \to h_{i+1}$ (consider radix 2^b):

$$c = h_i \gg b, \quad h_{i+1} = h_{i+1} + c, \quad h_i = h_i - c$$

Reduction (carries)

- Carry $h_i \to h_{i+1}$ (consider radix 2^b):

$$c = h_i \gg b, \quad h_{i+1} = h_{i+1} + c, \quad h_i = h_i - c$$

- Long carry chain:

$$h_0 \to h_1 \to h_2 \to h_3 \to h_4 \to h_5 \to h_6 \to h_7 \to h_8 \to h_9 \to h_0 \to h_1$$

(Drawback: suffering from long latency)

Reduction (carries)

- Carry $h_i \rightarrow h_{i+1}$ (consider radix 2^b):

$$c = h_i \gg b, \quad h_{i+1} = h_{i+1} + c, \quad h_i = h_i - c$$

- Long carry chain:

$$h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4 \rightarrow h_5 \rightarrow h_6 \rightarrow h_7 \rightarrow h_8 \rightarrow h_9 \rightarrow h_0 \rightarrow h_1$$

(Drawback: suffering from long latency)

- Reducing latency by interleaving carries:

$$h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4 \rightarrow h_5 \rightarrow h_6$$
$$h_5 \rightarrow h_6 \rightarrow h_7 \rightarrow h_8 \rightarrow h_9 \rightarrow h_0 \rightarrow h_1$$

Instruction-level optimization

instructions	ports	latency	reciprocal throughput
PADD/SUB(S,US)B/W/D/Q	p15	1	0.5
PMULDQ	p0	5	1
PSLLDQ, PSRLDQ	p5	1	1
PAND PANDN POR PXOR	p015	1	0.33

www.agner.org/optimize/instruction_tables.pdf

Instruction-level optimization

instructions	ports	latency	reciprocal throughput
PADD/SUB(S,US)B/W/D/Q	p15	1	0.5
PMULDQ	p0	5	1
PSLLDQ, PSRLDQ	p5	1	1
PAND PANDN POR PXOR	p015	1	0.33

www.agner.org/optimize/instruction_tables.pdf

- Each execution ports can handle 1 (micro-) instruction per cycle
- Useful for estimating the actual performance
  - PADD or PSLLDQ to multiply by 2?
- More on www.agner.org

Field inversion

- Used for converting from $(X : Y : Z)$ to $(X/Z, Y/Z)$, for example
  - typically only once

Field inversion

- Used for converting from $(X : Y : Z)$ to $(X/Z, Y/Z)$, for example
  - typically only once

- Extended Euclidean algorithm
  - $a, b \longmapsto x, y$ s.t. $xa + yb = \mathsf{gcd}(a, b)$
  - $a, p \longmapsto x, y$ s.t. $xa + yp = 1,$ i.e., $x \equiv a^{-1} \pmod p$
  - not (immediately) constant-time

Field inversion

- Used for converting from $(X : Y : Z)$ to $(X/Z, Y/Z)$, for example
  - typically only once
- Extended Euclidean algorithm
  - $a, b \longmapsto x, y$ s.t. $xa + yb = \text{gcd}(a, b)$
  - $a, p \longmapsto x, y$ s.t. $xa + yp = 1$, i.e., $x \equiv a^{-1} \pmod{p}$
  - not (immediately) constant-time
- Square-and-multiply
  - $a^{p-1} \equiv 1 \pmod{p} \Rightarrow a^{p-2}$ is the inverse!

Field inversion

- Used for converting from $(X : Y : Z)$ to $(X/Z, Y/Z)$, for example
    - typically only once

- Extended Euclidean algorithm
    - $a, b \longmapsto x, y$ s.t. $xa + yb = \text{gcd}(a, b)$
    - $a, p \longmapsto x, y$ s.t. $xa + yp = 1,$ i.e., $x \equiv a^{-1} \pmod{p}$
    - not (immediately) constant-time

- Square-and-multiply
    - $a^{p-1} \equiv 1 \pmod{p} \Rightarrow a^{p-2}$ is the inverse!
    - $p - 2$ is public

SageMath


RSS · Blog · Trac · Wiki · Questions? · Donate
Online: CoCalc · SageCell or Download, Source Code
v8.4 (2018-10-17) ·
Language
Home Tour Help Library Download Development Links
SageMath is a free open-source mathematics software system licensed under the GPL. It builds on top of many existing open-source packages: NumPy, SciPy, matplotlib, Sympy, Maxima, GAP, FLINT, R and many more. Access their combined power through a common, Python-based language or directly via interfaces or wrappers.
Mission: Creating a viable free open source alternative to Magma, Maple, Mathematica and Matlab.
Do you want to learn how to use SageMath?
Read Sage for Undergraduates by Gregory Bard or
Mathematical Computation with Sage by Paul Zimmermann et al.
translations: Calcul mathématique avec Sage (French), Rechnen mit Sage (German)
CoCalc (SageMathCloud)
or: SageMathCell
Download 8.4
Changelogs · Source 8.4 · Packages · Git
Help/Documentation
Video · Forums · Tutorial · FAQ · Questions?
Feature Tour
Quickstart · Research · Graphics
Library
Testimonials · Books · Publications · Press Kit
Search
Search...

doc.sagemath.org/html/en/constructions/elliptic_curves.html

Sage Constructions v8.4 »

Table Of Contents

Previous topic

Modular forms

Next topic

Number fields

This Page

Show Source

Quick search

Go

Elliptic curves

Conductor

How do you compute the conductor of an elliptic curve (over $\mathbf{Q}$) in Sage?

Once you define an elliptic curve E in Sage, using the `EllipticCurve` command, the conductor is one of several "methods" associated to E. Here is an example of the syntax (borrowed from section 2.4 "Modular forms" in the tutorial):

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over
Rational Field
sage: E.conductor()
10351
```

j-invariant

How do you compute the j-invariant of an elliptic curve in Sage?

Other methods associated to the `EllipticCurve` class are `j_invariant`, `discriminant`, and `weierstrass_model`. Here is an example of their syntax.

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: E.j_invariant()
-122023936/161051
sage: E.short_weierstrass_model()
Elliptic Curve defined by y^2  = x^3 - 13392*x - 1080432 over Rational Field
sage: E.discriminant()
-161051
sage: E = EllipticCurve(GF(5),[0, -1, 1, -10, -20])
sage: E.short_weierstrass_model()
Elliptic Curve defined by y^2  = x^3 + 3*x + 3 over Finite Field of size 5
sage: E.j_invariant()
4
```