Software implementation of ECC

Radboud University, Nijmegen, The Netherlands

June 4, 2015

Summer school on real-world crypto and privacy
Šibenik, Croatia

Software implementation of (H)ECC

Radboud University, Nijmegen, The Netherlands

June 4, 2015

Summer school on real-world crypto and privacy
Šibenik, Croatia

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 $kP = Q$.

Efficient ECC

- First idea: user needs to compute kP, so make that fast

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 $kP = Q$.

Efficient ECC

- First idea: user needs to compute kP, so make that fast
- Actual situation is more complex:
  - Keypair generation: Compute kP for **fixed** P,
    don't leak information about scalar k

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 $kP = Q$.

Efficient ECC

- First idea: user needs to compute kP, so make that fast
- Actual situation is more complex:
  - Keypair generation: Compute kP for **fixed** P,
    don't leak information about scalar k
  - DH common-key computation: Compute kP for **variable** P,
    don't leak information about scalar k

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 $kP = Q$.

Efficient ECC

- First idea: user needs to compute kP, so make that fast
- Actual situation is more complex:
  - Keypair generation: Compute kP for **fixed** P,
    don't leak information about scalar k
  - DH common-key computation: Compute kP for **variable** P,
    don't leak information about scalar k
  - Signature verification needs $k_1P_1 + k_2P_2$,
    ok to leak information about (public) scalars k_1 and k_2

The ECDLP

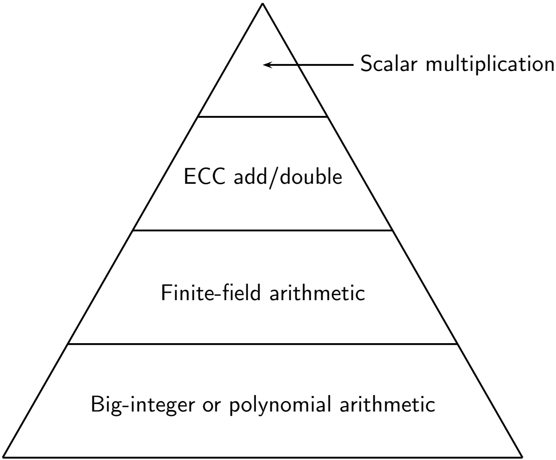
Definition

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

Efficient ECC

- First idea: user needs to compute kP, so make that fast
- Actual situation is more complex:
  - Keypair generation: Compute kP for **fixed** P, **don't leak** information about scalar k
  - DH common-key computation: Compute kP for **variable** P, **don't leak** information about scalar k
  - Signature verification needs $k_1P_1 + k_2P_2$, **ok to leak** information about (public) scalars k_1 and k_2

The ECC implementation pyramid



Why I don't like the pyramid...

- Pyramid levels are *not* independent
- Interactions through all levels, relevant for
  - Correctness,
  - Security, and
  - Performance

Why I don't like the pyramid. . .

- Pyramid levels are *not* independent
- Interactions through all levels, relevant for
  - Correctness,
  - Security, and
  - Performance
- Plan for today: demonstrate these dependencies

Why I don't like the pyramid...

- Pyramid levels are *not* independent
- Interactions through all levels, relevant for
  - Correctness,
  - Security, and
  - Performance
- Plan for today: demonstrate these dependencies
- Fix target architecture: AMD64 (aka x86_64, aka x64)
- Fix target microarchitecture: Intel Sandy Bridge and Ivy Bridge

Let's start with 255-bit integers

```
typedef struct{
  unsigned long long a[4];
}  bigint255;

void bigint255_add(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  r->a[0] = x->a[0] + y->a[0];
  r->a[1] = x->a[1] + y->a[1];
  r->a[2] = x->a[2] + y->a[2];
  r->a[3] = x->a[3] + y->a[3];
}
```

Let's start with 255-bit integers

```
typedef struct{
  unsigned long long a[4];
}  bigint255;

void bigint255_add(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  r->a[0] = x->a[0] + y->a[0];
  r->a[1] = x->a[1] + y->a[1];
  r->a[2] = x->a[2] + y->a[2];
  r->a[3] = x->a[3] + y->a[3];
}
```

- What's wrong about this?

Let's start with 255-bit integers

```
typedef struct{
  unsigned long long a[4];
}  bigint255;

void bigint255_add(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  r->a[0] = x->a[0] + y->a[0];
  r->a[1] = x->a[1] + y->a[1];
  r->a[2] = x->a[2] + y->a[2];
  r->a[3] = x->a[3] + y->a[3];
}
```

- What's wrong about this?
- This performs arithmetic on a vector of 4 independent 64-bit integers (modulo 2^{64})

Let's start with 255-bit integers

```
typedef struct{
  unsigned long long a[4];
}  bigint255;

void bigint255_add(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  r->a[0] = x->a[0] + y->a[0];
  r->a[1] = x->a[1] + y->a[1];
  r->a[2] = x->a[2] + y->a[2];
  r->a[3] = x->a[3] + y->a[3];
}
```

- What's wrong about this?
- This performs arithmetic on a vector of 4 independent 64-bit integers (modulo 2^{64})
- This is *not* the same as arithmetic on 256-bit integers
- Need to ripple the carries of all additions!

Radix-2^{51} representation

- Radix-2^{64} representation works and is sometimes a good choice
- Highly depends on the efficiency of handling carries

Radix-2^{51} representation

- Radix-2^{64} representation works and is sometimes a good choice
- Highly depends on the efficiency of handling carries
- Example 1: Intel Nehalem can do 3 additions every cycle, but only 1 addition with carry every two cycles (carries cost a factor of 6!)

Radix-2^{51} representation

- Radix-2^{64} representation works and is sometimes a good choice
- Highly depends on the efficiency of handling carries
- Example 1: Intel Nehalem can do 3 additions every cycle, but only 1 addition with carry every two cycles (carries cost a factor of 6!)
- Example 2: When using vector arithmetic, carries are typically lost (expensive to recompute)

Radix-2^{51} representation

- Radix-2^{64} representation works and is sometimes a good choice
- Highly depends on the efficiency of handling carries
- Example 1: Intel Nehalem can do 3 additions every cycle, but only 1 addition with carry every two cycles (carries cost a factor of 6!)
- Example 2: When using vector arithmetic, carries are typically lost (expensive to recompute)
- Let's get rid of the carries, represent A as $(a_0, a_1, a_2, a_3, a_4)$ with

$$A = \sum_{i=0}^{4} a_i 2^{51 \cdot i}$$

- This is called radix-2^{51} representation

Radix-2^{51} representation

- Radix-2^{64} representation works and is sometimes a good choice
- Highly depends on the efficiency of handling carries
- Example 1: Intel Nehalem can do 3 additions every cycle, but only 1 addition with carry every two cycles (carries cost a factor of 6!)
- Example 2: When using vector arithmetic, carries are typically lost (expensive to recompute)
- Let's get rid of the carries, represent A as $(a_0, a_1, a_2, a_3, a_4)$ with

$$A = \sum_{i=0}^{4} a_i 2^{51 \cdot i}$$

- This is called radix-2^{51} representation
- Multiple ways to write the same integer A, for example $A = 2^{52}$:
  - $(2^{52}, 0, 0, 0, 0)$
  - $(0, 2, 0, 0, 0)$

Addition of two bigint255

```
typedef struct{
  unsigned long long a[5];
}  bigint255;

void bigint255_add(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  r->a[0] = x->a[0] + y->a[0];
  r->a[1] = x->a[1] + y->a[1];
  r->a[2] = x->a[2] + y->a[2];
  r->a[3] = x->a[3] + y->a[3];
  r->a[4] = x->a[4] + y->a[4];
}
```

Addition of two bigint255

```
typedef struct{
  unsigned long long a[5];
}  bigint255;

void bigint255_add(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  r->a[0] = x->a[0] + y->a[0];
  r->a[1] = x->a[1] + y->a[1];
  r->a[2] = x->a[2] + y->a[2];
  r->a[3] = x->a[3] + y->a[3];
  r->a[4] = x->a[4] + y->a[4];
}
```

Addition of two bigint255

```
typedef struct{
  unsigned long long a[5];
}  bigint255;

void bigint255_add(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  r->a[0] = x->a[0] + y->a[0];
  r->a[1] = x->a[1] + y->a[1];
  r->a[2] = x->a[2] + y->a[2];
  r->a[3] = x->a[3] + y->a[3];
  r->a[4] = x->a[4] + y->a[4];
}
```

- This works as long as all coefficients are in $[0, \ldots, 2^{63} - 1]$

Addition of two `bigint255`

```
typedef struct{
  unsigned long long a[5];
}  bigint255;

void bigint255_add(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  r->a[0] = x->a[0] + y->a[0];
  r->a[1] = x->a[1] + y->a[1];
  r->a[2] = x->a[2] + y->a[2];
  r->a[3] = x->a[3] + y->a[3];
  r->a[4] = x->a[4] + y->a[4];
}
```

- This works as long as all coefficients are in $[0, \ldots, 2^{63} - 1]$
- When starting with 51-bit coefficients, we can do quite a few additions before we have to carry

Subtraction of two `bigint255`

```
typedef struct{
  signed long long a[5];
}  bigint255;

void bigint255_sub(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  r->a[0] = x->a[0] - y->a[0];
  r->a[1] = x->a[1] - y->a[1];
  r->a[2] = x->a[2] - y->a[2];
  r->a[3] = x->a[3] - y->a[3];
  r->a[4] = x->a[4] - y->a[4];
}
```

- Slightly update our `bigint255` definition to work with *signed* 64-bit integers

Carrying in radix-2^{51}

- With many additions, coefficients may grow larger than 63 bits
- They grow even faster in multiplication

Carrying in radix-2^{51}

- With many additions, coefficients may grow larger than 63 bits
- They grow even faster in multiplication
- Eventually we have to *carry* en bloc:

```
signed long long carry = r.a[0] >> 51;
r.a[1] += carry;
carry <<= 51;
r.a[0] -= carry;
```

Carrying in radix-2^{51}

- With many additions, coefficients may grow larger than 63 bits
- They grow even faster in multiplication
- Eventually we have to *carry* en bloc:

```
signed long long carry = r.a[0] >> 51;
r.a[1] += carry;
carry <<= 51;
r.a[0] -= carry;
```

- Similar for all higher coefficients...

Big integers and polynomials

- Addition/Subtraction code would look *exactly* the same for 5-coefficient polynomial addition

Big integers and polynomials

- Addition/Subtraction code would look *exactly* the same for 5-coefficient polynomial addition
- This is no coincidence: We actually perform arithmetic in $\mathbb{Z}[x]$
- Inputs to addition/subtraction are 5-coefficient polynomials

Big integers and polynomials

- Addition/Subtraction code would look *exactly* the same for 5-coefficient polynomial addition
- This is no coincidence: We actually perform arithmetic in $\mathbb{Z}[x]$
- Inputs to addition/subtraction are 5-coefficient polynomials
- Nice thing about arithmetic $\mathbb{Z}[x]$: no carries!

Big integers and polynomials

- Addition/Subtraction code would look *exactly* the same for 5-coefficient polynomial addition
- This is no coincidence: We actually perform arithmetic in $\mathbb{Z}[x]$
- Inputs to addition/subtraction are 5-coefficient polynomials
- Nice thing about arithmetic $\mathbb{Z}[x]$: no carries!
- To go from $\mathbb{Z}[x]$ to $\mathbb{Z}$, evaluate at the radix (this is a ring homomorphism)
- Carrying means evaluating at the radix

Big integers and polynomials

- Addition/Subtraction code would look *exactly* the same for 5-coefficient polynomial addition
- This is no coincidence: We actually perform arithmetic in $\mathbb{Z}[x]$
- Inputs to addition/subtraction are 5-coefficient polynomials
- Nice thing about arithmetic $\mathbb{Z}[x]$: no carries!
- To go from $\mathbb{Z}[x]$ to $\mathbb{Z}$, evaluate at the radix (this is a ring homomorphism)
- Carrying means evaluating at the radix
- Thinking of multiprecision integers as polynomials is very powerful for efficient arithmetic

Using floating-point limbs

- Now we can also use floats for our coefficients
- An IEEE-754 floating-point number has value

$$(-1)^s \cdot (1.b_{m-1}b_{m-2}\dots b_0) \cdot 2^{e-t} \text{ with } b_i \in \{0,1\}$$

Using floating-point limbs

- Now we can also use floats for our coefficients
- An IEEE-754 floating-point number has value

$$(-1)^s \cdot (1.b_{m-1}b_{m-2}\dots b_0) \cdot 2^{e-t} \text{ with } b_i \in \{0,1\}$$

- For double-precision floats:
  - $s \in \{0,1\}$ "sign bit"
  - $m = 52$ "mantissa bits"
  - $e \in \{1,\dots,2046\}$ "exponent"
  - $t = 1023$

Using floating-point limbs

- Now we can also use floats for our coefficients
- An IEEE-754 floating-point number has value

$$(-1)^s \cdot (1.b_{m-1}b_{m-2}\dots b_0) \cdot 2^{e-t} \text{ with } b_i \in \{0,1\}$$

- For double-precision floats:
  - $s \in \{0,1\}$ "sign bit"
  - $m = 52$ "mantissa bits"
  - $e \in \{1,\dots,2046\}$ "exponent"
  - $t = 1023$
- For single-precision floats:
  - $s \in \{0,1\}$ "sign bit"
  - $m = 23$ "mantissa bits"
  - $e \in \{1,\dots,254\}$ "exponent"
  - $t = 127$

Using floating-point limbs

- Now we can also use floats for our coefficients
- An IEEE-754 floating-point number has value

$$(-1)^s \cdot (1.b_{m-1}b_{m-2}\dots b_0) \cdot 2^{e-t} \text{ with } b_i \in \{0,1\}$$

- For double-precision floats:
  - $s \in \{0,1\}$ "sign bit"
  - $m = 52$ "mantissa bits"
  - $e \in \{1,\dots,2046\}$ "exponent"
  - $t = 1023$
- For single-precision floats:
  - $s \in \{0,1\}$ "sign bit"
  - $m = 23$ "mantissa bits"
  - $e \in \{1,\dots,254\}$ "exponent"
  - $t = 127$
- Exponent $= 0$ used to represent 0

Using floating-point limbs

- Now we can also use floats for our coefficients
- An IEEE-754 floating-point number has value

$$(-1)^s \cdot (1.b_{m-1}b_{m-2}\dots b_0) \cdot 2^{e-t} \text{ with } b_i \in \{0,1\}$$

- For double-precision floats:
  - $s \in \{0,1\}$ "sign bit"
  - $m = 52$ "mantissa bits"
  - $e \in \{1,\dots,2046\}$ "exponent"
  - $t = 1023$
- For single-precision floats:
  - $s \in \{0,1\}$ "sign bit"
  - $m = 23$ "mantissa bits"
  - $e \in \{1,\dots,254\}$ "exponent"
  - $t = 127$
- Exponent $= 0$ used to represent 0
- Any number that can be represented like this, will be precise
- Other numbers will be *rounded*, according to a rounding mode

Addition

```
typedef struct{
  double a[12];
}  bigint255;

void bigint255_add(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  int i;
  for(i=0;i<12;i++)
    r->a[i] = x->a[i] + y->a[i];
}
```

Subtraction

```
typedef struct{
  double a[12];
} bigint255;

void bigint255_sub(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  int i;
  for(i=0;i<12;i++)
    r->a[i] = x->a[i] - y->a[i];
}
```

Carrying

- For carrying integers we used a right shift (discard lowest bits)

Carrying

- For carrying integers we used a right shift (discard lowest bits)
- For floating-point numbers we can use multiplication by the inverse of the radix
- Example: Radix 2^{22}, multiply by 2^{-22}
- This does *not* cut off lowest bits, need to round

Carrying

- For carrying integers we used a right shift (discard lowest bits)
- For floating-point numbers we can use multiplication by the inverse of the radix
- Example: Radix 2^{22}, multiply by 2^{-22}
- This does *not* cut off lowest bits, need to round
- Some processors have efficient rounding instructions, e.g., `vroundpd`

Carrying

- For carrying integers we used a right shift (discard lowest bits)
- For floating-point numbers we can use multiplication by the inverse of the radix
- Example: Radix 2^{22}, multiply by 2^{-22}
- This does *not* cut off lowest bits, need to round
- Some processors have efficient rounding instructions, e.g., `vroundpd`
- Otherwise (for double-precision):
  - add constant $2^{52} + 2^{51}$
  - subtract constant $2^{52} + 2^{51}$
  - This will round the number to an integer according to the rounding mode (to nearest, towards zero, away from zero, or truncate)

Why would you want this?

- ECC is typically bottlenecked by speed of multiplier
- Intel Sandy Bridge, Ivy Bridge:
  - One $64 \times 64 \to 128$ multiplication per cycle

Why would you want this?

- ECC is typically bottlenecked by speed of multiplier
- Intel Sandy Bridge, Ivy Bridge:
  - One $64 \times 64 \to 128$ multiplication per cycle
  - Four (vectorized) double-precision multiplications per cycle

Why would you want this?

- ECC is typically bottlenecked by speed of multiplier
- Intel Sandy Bridge, Ivy Bridge:
  - One $64 \times 64 \to 128$ multiplication per cycle
  - Four (vectorized) double-precision multiplications per cycle
  - Four (vectorized) double-precision additions in the same cycle

Why would you want this?

- ECC is typically bottlenecked by speed of multiplier
- Intel Sandy Bridge, Ivy Bridge:
  - One $64 \times 64 \rightarrow 128$ multiplication per cycle
  - Four (vectorized) double-precision multiplications per cycle
  - Four (vectorized) double-precision additions in the same cycle
- Operations on 256-bit vector registers introduced with AVX

Why would you want this?

- ECC is typically bottlenecked by speed of multiplier
- Intel Sandy Bridge, Ivy Bridge:
  - One $64 \times 64 \to 128$ multiplication per cycle
  - Four (vectorized) double-precision multiplications per cycle
  - Four (vectorized) double-precision additions in the same cycle
- Operations on 256-bit vector registers introduced with AVX
- *Integer* operations on those registers introduced only with AVX2
- Sandy Bridge and Ivy Bridge don't have AVX2

Vectorizing EC scalar multiplication

Computing multiple scalar multiplications

- Changes the rules of the game
- Increases size of active data set

Vectorizing EC scalar multiplication

Computing multiple scalar multiplications

- Changes the rules of the game
- Increases size of active data set

Parallelism inside multiprecision arithmetic

- Addition (in redundant representation) is trivially vectorized
- Vectorizing multiplication needs many shuffles
- Vectorization "eats up" instruction-level parallelism

Vectorizing EC scalar multiplication

Computing multiple scalar multiplications

- Changes the rules of the game
- Increases size of active data set

Parallelism inside multiprecision arithmetic

- Addition (in redundant representation) is trivially vectorized
- Vectorizing multiplication needs many shuffles
- Vectorization “eats up” instruction-level parallelism

Parallelism inside EC arithmetic

- Vectorize independent multiplications in EC addition
- May still need some shuffles (after each block of operations)
- Efficiency depends on EC formulas

Example: Montgomery ladder step

function ladderstep($x_{Q-P}, X_P, Z_P, X_Q, Z_Q$)

$t_1 \leftarrow X_P + Z_P$

$t_6 \leftarrow t_1^2$

$t_2 \leftarrow X_P - Z_P$

$t_7 \leftarrow t_2^2$

$t_5 \leftarrow t_6 - t_7$

$t_3 \leftarrow X_Q + Z_Q$

$t_4 \leftarrow X_Q - Z_Q$

$t_8 \leftarrow t_4 \cdot t_1$

$t_9 \leftarrow t_3 \cdot t_2$

$X_{P+Q} \leftarrow (t_8 + t_9)^2$

$Z_{P+Q} \leftarrow x_{Q-P} \cdot (t_8 - t_9)^2$

$X_{[2]P} \leftarrow t_6 \cdot t_7$

$Z_{[2]P} \leftarrow t_5 \cdot (t_7 + ((A+2)/4) \cdot t_5)$

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

end function

Example: Montgomery ladder step

function ladderstep($x_{Q-P}, X_P, Z_P, X_Q, Z_Q$)

$t_1 \leftarrow X_P + Z_P; t_2 \leftarrow X_P - Z_P; t_3 \leftarrow X_Q + Z_Q; t_4 \leftarrow X_Q - Z_Q$

$t_6 \leftarrow t_1 \cdot t_1; t_7 \leftarrow t_2 \cdot t_2; t_8 \leftarrow t_4 \cdot t_1; t_9 \leftarrow t_3 \cdot t_2$

$t_{10} \leftarrow ((A+2)/4) \cdot t_6$
$t_{11} \leftarrow ((A+2)/4 - 1) \cdot t_7$

$t_5 \leftarrow t_6 - t_7; t_4 \leftarrow t_{10} - t_{11}; t_1 \leftarrow t_8 - t_9; t_0 \leftarrow t_8 + t_9$

$Z_{[2]P} \leftarrow t_5 \cdot t_4; X_{P+Q} \leftarrow t_0^2; X_{[2]P} \leftarrow t_6 \cdot t_7; t_2 \leftarrow t_1 \cdot t_1$

$Z_{P+Q} \leftarrow x_{Q-P} \cdot t_2$

return $(X_{[2]P}, Z_{[2]P}, X_{P+Q}, Z_{P+Q})$
end function

A better candidate: Kummer surfaces

- Think of a Kummer surface as the Jacobian of a hyperelliptic curve modulo negation

A better candidate: Kummer surfaces

- Think of a Kummer surface as the Jacobian of a hyperelliptic curve modulo negation
- Easier way to think about it:
  - Group modulo negation
  - Map from group to Kummer surface by rational map X
  - Elements represented projectively as $(x : y : z : t)$
  - $(x : y : z : t) = (rx : ry : rz : rt)$ for any $r \neq 0$
  - Efficient doubling and efficient *differential addition*

A better candidate: Kummer surfaces

- Think of a Kummer surface as the Jacobian of a hyperelliptic curve modulo negation
- Easier way to think about it:
  - Group modulo negation
  - Map from group to Kummer surface by rational map X
  - Elements represented projectively as $(x : y : z : t)$
  - $(x : y : z : t) = (rx : ry : rz : rt)$ for any $r \neq 0$
  - Efficient doubling and efficient *differential addition*
- Ladderstep: gets as input $X(P) = (x_2 : y_2 : z_2 : t_2)$, $X(Q) = (x_3 : y_3 : z_3 : t_3)$, and $X(Q - P) = (x_1 : y_1 : z_1 : t_1)$
  - Computes $X(2P) = (x_4 : y_4 : z_4 : t_4)$
  - Computes $X(P + Q) = (x_5 : y_5 : z_5 : t_5)$

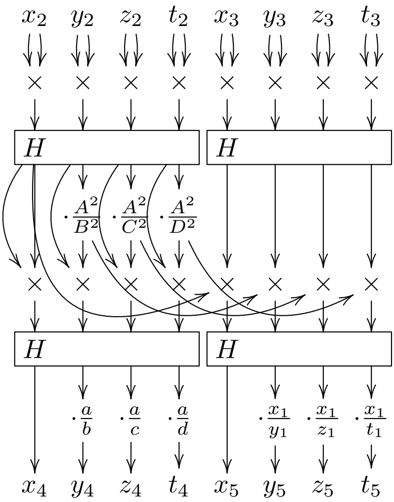
A better candidate: Kummer surfaces

- Think of a Kummer surface as the Jacobian of a hyperelliptic curve modulo negation
- Easier way to think about it:
  - Group modulo negation
  - Map from group to Kummer surface by rational map X
  - Elements represented projectively as $(x : y : z : t)$
  - $(x : y : z : t) = (rx : ry : rz : rt)$ for any $r \neq 0$
  - Efficient doubling and efficient *differential addition*
- Ladderstep: gets as input $X(P) = (x_2 : y_2 : z_2 : t_2)$, $X(Q) = (x_3 : y_3 : z_3 : t_3)$, and $X(Q - P) = (x_1 : y_1 : z_1 : t_1)$
  - Computes $X(2P) = (x_4 : y_4 : z_4 : t_4)$
  - Computes $X(P + Q) = (x_5 : y_5 : z_5 : t_5)$
- Coordinates are elements of a (large) finite field

A better candidate: Kummer surfaces

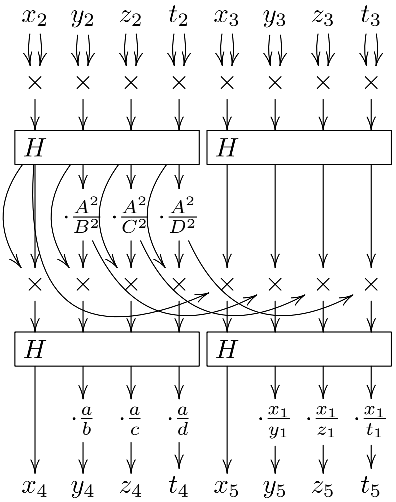
- Think of a Kummer surface as the Jacobian of a hyperelliptic curve modulo negation
- Easier way to think about it:
  - Group modulo negation
  - Map from group to Kummer surface by rational map X
  - Elements represented projectively as $(x : y : z : t)$
  - $(x : y : z : t) = (rx : ry : rz : rt)$ for any $r \neq 0$
  - Efficient doubling and efficient *differential addition*
- Ladderstep: gets as input $X(P) = (x_2 : y_2 : z_2 : t_2)$, $X(Q) = (x_3 : y_3 : z_3 : t_3)$, and $X(Q - P) = (x_1 : y_1 : z_1 : t_1)$
  - Computes $X(2P) = (x_4 : y_4 : z_4 : t_4)$
  - Computes $X(P + Q) = (x_5 : y_5 : z_5 : t_5)$
- Coordinates are elements of a (large) finite field
- For same security level, underlying field has half the size as for ECC
- Example: Choose ≈ 128-bit field for ≈ 128 bits of security

Arithmetic on the Kummer surface

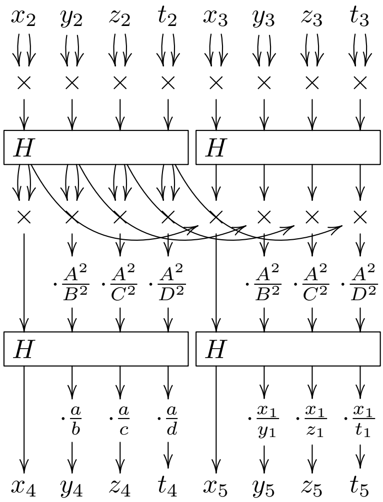


$10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ ladder formulas

Arithmetic on the Kummer surface



$10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ ladder formulas

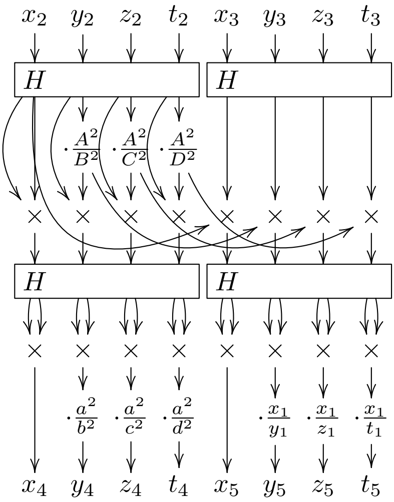


$7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ ladder formulas

The “squared Kummer surface”

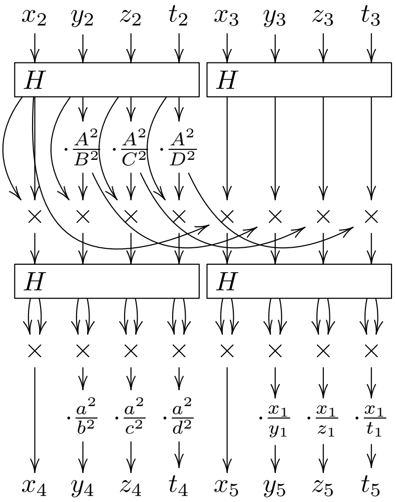
- In fact, we use arithmetic on a different, “squared” surface
- Each point $(x : y : z : t)$ on the original surface corresponds to $(x^2 : y^2 : z^2 : t^2)$ on the squared surface
- No operation-count advantages
- Easier to construct squared surface with small constants
- In the following rename $(x^2 : y^2 : z^2 : t^2)$ to $(x : y : z : t)$

Arithmetic on the squared Kummer surface

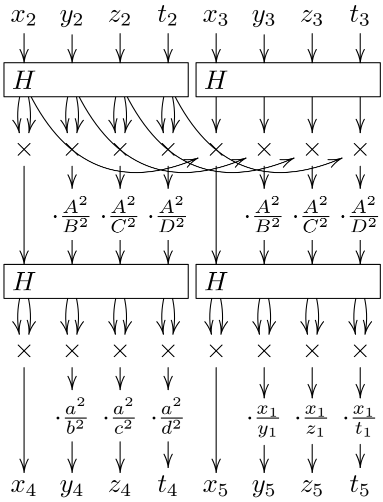


$10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ ladder formulas

Arithmetic on the squared Kummer surface

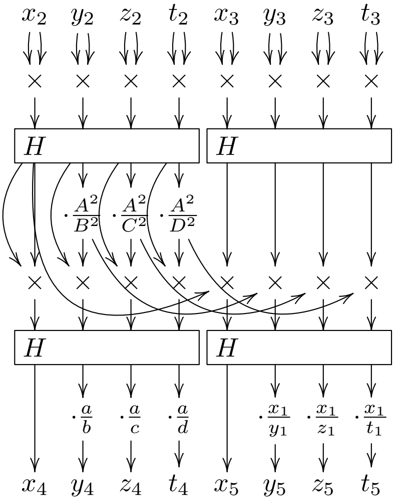


$10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ ladder formulas

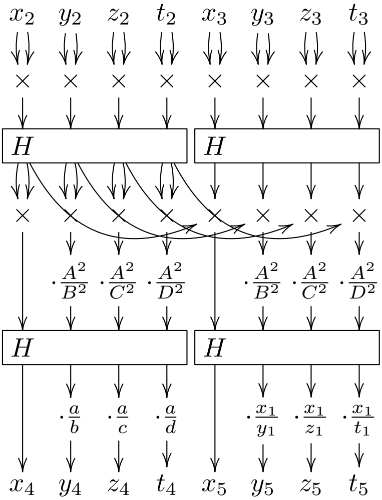


$7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ ladder formulas

Arithmetic on the (original) Kummer surface



$10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ ladder formulas



$7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ ladder formulas

A suitable Kummer surface

- Formulas for efficient Kummer surface arithmetic known for a while
  - Originally proposed by Chudnovsky, Chudnovsky, 1986
  - $10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ formulas by Gaudry, 2006
  - $7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ formulas by Bernstein, 2006

A suitable Kummer surface

- Formulas for efficient Kummer surface arithmetic known for a while
  - Originally proposed by Chudnovsky, Chudnovsky, 1986
  - $10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ formulas by Gaudry, 2006
  - $7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ formulas by Bernstein, 2006
- Problem: find cryptographically secure surface with small constants

A suitable Kummer surface

- Formulas for efficient Kummer surface arithmetic known for a while
  - Originally proposed by Chudnovsky, Chudnovsky, 1986
  - $10\mathbf{M}+9\mathbf{S}+6\mathbf{m}$ formulas by Gaudry, 2006
  - $7\mathbf{M}+12\mathbf{S}+9\mathbf{m}$ formulas by Bernstein, 2006
- Problem: find cryptographically secure surface with small constants
- Gaudry, Schost, 2012: suitable (squared) surface:
  - Defined over the field $\mathbb{F}_{2^{127}-1}$
  - $(1 : a^2/b^2 : a^2/c^2 : a^2/d^2) = (-114 : 57 : 66 : 418)$
  - $(1 : A^2/B^2 : A^2/C^2 : A^2/D^2) = (-833 : 2499 : 1617 : 561)$

A suitable Kummer surface

- Formulas for efficient Kummer surface arithmetic known for a while
  - Originally proposed by Chudnovsky, Chudnovsky, 1986
  - $10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ formulas by Gaudry, 2006
  - $7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ formulas by Bernstein, 2006
- Problem: find cryptographically secure surface with small constants
- Gaudry, Schost, 2012: suitable (squared) surface:
  - Defined over the field $\mathbb{F}_{2^{127}-1}$
  - $(1 : a^2/b^2 : a^2/c^2 : a^2/d^2) = (-114 : 57 : 66 : 418)$
  - $(1 : A^2/B^2 : A^2/C^2 : A^2/D^2) = (-833 : 2499 : 1617 : 561)$
- Finding this surface cost $1\,000\,000$ CPU hours
- The same surface has been used by Bos, Costello, Hisil, and Lauter (Eurocrypt 2013)

Representing elements of $\mathbb{F}_{2^{127}-1}$

- Represent an element A in radix-$2^{127/6}$
- Write A as $a_0, a_1, a_2, a_3, a_4, a_5$, where
  - a_0 is a small multiple of 2^0
  - a_1 is a small multiple of 2^{22}
  - a_2 is a small multiple of 2^{43}
  - a_3 is a small multiple of 2^{64}
  - a_4 is a small multiple of 2^{85}
  - a_5 is a small multiple of 2^{106}

Multiplication

- Consider multiplication of A and B with reduction mod $2^{127} - 1$
- Make use of the fact that $2^{127} \equiv 1$
- With radix $2^{127/6}$ we obtain:

$$
\begin{aligned}
r_0 &= a_0b_0 + 2^{-127}a_1b_5 + 2^{-127}a_2b_4 + 2^{-127}a_3b_3 + 2^{-127}a_4b_2 + 2^{-127}a_5b_1 \\
r_1 &= a_0b_1 + a_1b_0 + 2^{-127}a_2b_5 + 2^{-127}a_3b_4 + 2^{-127}a_4b_3 + 2^{-127}a_5b_2 \\
r_2 &= a_0b_2 + a_1b_1 + a_2b_0 + 2^{-127}a_3b_5 + 2^{-127}a_4b_4 + 2^{-127}a_5b_3 \\
r_3 &= a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + 2^{-127}a_4b_5 + 2^{-127}a_5b_4 \\
r_4 &= a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 + 2^{-127}a_5b_5 \\
r_5 &= a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0
\end{aligned}
$$

Multiplication

- Consider multiplication of A and B with reduction mod $2^{127} - 1$
- Make use of the fact that $2^{127} \equiv 1$
- With radix $2^{127/6}$ we obtain:

$$\begin{aligned}
r_0 &= a_0b_0 + 2^{-127}a_1b_5 + 2^{-127}a_2b_4 + 2^{-127}a_3b_3 + 2^{-127}a_4b_2 + 2^{-127}a_5b_1 \\
r_1 &= a_0b_1 + a_1b_0 + 2^{-127}a_2b_5 + 2^{-127}a_3b_4 + 2^{-127}a_4b_3 + 2^{-127}a_5b_2 \\
r_2 &= a_0b_2 + a_1b_1 + a_2b_0 + 2^{-127}a_3b_5 + 2^{-127}a_4b_4 + 2^{-127}a_5b_3 \\
r_3 &= a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + 2^{-127}a_4b_5 + 2^{-127}a_5b_4 \\
r_4 &= a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 + 2^{-127}a_5b_5 \\
r_5 &= a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0
\end{aligned}$$

- Obviously, we always perform this whole thing $4\times$ in parallel

Multiplication

- Consider multiplication of A and B with reduction mod $2^{127} - 1$
- Make use of the fact that $2^{127} \equiv 1$
- With radix $2^{127/6}$ we obtain:

$$
\begin{aligned}
r_0 &= a_0b_0 + 2^{-127}a_1b_5 + 2^{-127}a_2b_4 + 2^{-127}a_3b_3 + 2^{-127}a_4b_2 + 2^{-127}a_5b_1 \\
r_1 &= a_0b_1 + a_1b_0 + 2^{-127}a_2b_5 + 2^{-127}a_3b_4 + 2^{-127}a_4b_3 + 2^{-127}a_5b_2 \\
r_2 &= a_0b_2 + a_1b_1 + a_2b_0 + 2^{-127}a_3b_5 + 2^{-127}a_4b_4 + 2^{-127}a_5b_3 \\
r_3 &= a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + 2^{-127}a_4b_5 + 2^{-127}a_5b_4 \\
r_4 &= a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 + 2^{-127}a_5b_5 \\
r_5 &= a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0
\end{aligned}
$$

- Obviously, we always perform this whole thing $4\times$ in parallel
- Obviously, we specialize squaring

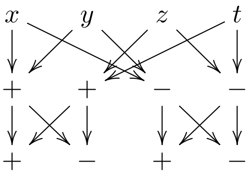
Multiplication

- Consider multiplication of A and B with reduction mod $2^{127}-1$
- Make use of the fact that $2^{127} \equiv 1$
- With radix $2^{127/6}$ we obtain:

$$
\begin{aligned}
r_0 &= a_0b_0 + 2^{-127}a_1b_5 + 2^{-127}a_2b_4 + 2^{-127}a_3b_3 + 2^{-127}a_4b_2 + 2^{-127}a_5b_1 \\
r_1 &= a_0b_1 + a_1b_0 + 2^{-127}a_2b_5 + 2^{-127}a_3b_4 + 2^{-127}a_4b_3 + 2^{-127}a_5b_2 \\
r_2 &= a_0b_2 + a_1b_1 + a_2b_0 + 2^{-127}a_3b_5 + 2^{-127}a_4b_4 + 2^{-127}a_5b_3 \\
r_3 &= a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + 2^{-127}a_4b_5 + 2^{-127}a_5b_4 \\
r_4 &= a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 + 2^{-127}a_5b_5 \\
r_5 &= a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0
\end{aligned}
$$

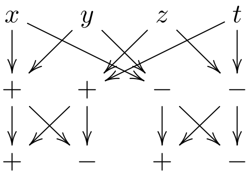
- Obviously, we always perform this whole thing $4\times$ in parallel
- Obviously, we specialize squaring
- Obviously, we specialize multiplications by small constants

The Hadamard transform



- Only shuffeling operation in Kummer arithmetic
- AVX has limited shuffeling across left and right half
- Plain Hadamard turns out to be expensive

The Hadamard transform

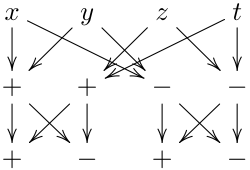


- Only shuffeling operation in Kummer arithmetic
- AVX has limited shuffeling across left and right half
- Plain Hadamard turns out to be expensive

Permuted and negated Hadamard

- Allow generalized Hadamard to output permuted vector
- Self-inverting permutation "cleans" after two generalized Hadamards

The Hadamard transform

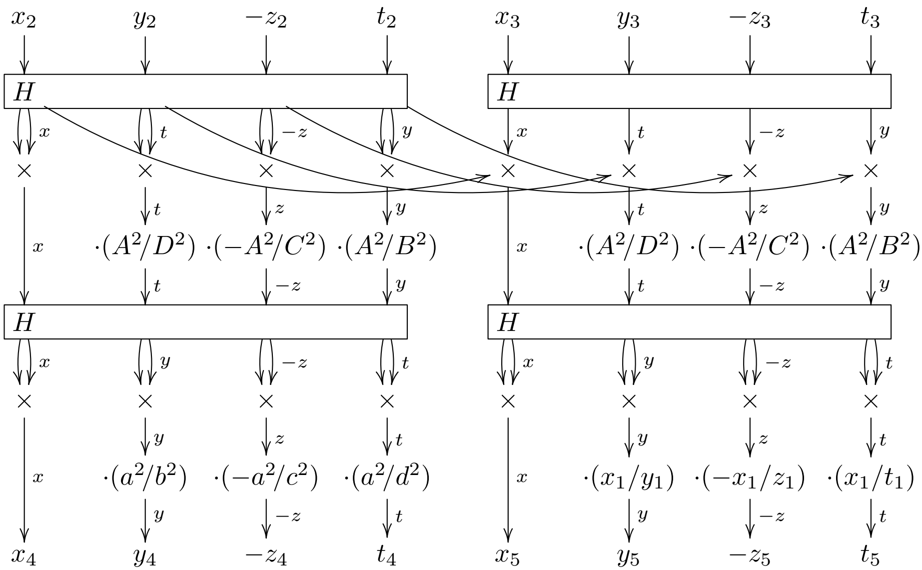


- Only shuffeling operation in Kummer arithmetic
- AVX has limited shuffeling across left and right half
- Plain Hadamard turns out to be expensive

Permuted and negated Hadamard

- Allow generalized Hadamard to output permuted vector
- Self-inverting permutation "cleans" after two generalized Hadamards
- Allow generalized Hadamard to negate vector entries
- "Clean" negations by multiplication by negated constants

Arithmetic on the squared Kummer surface



Looking back...

- Fastest computation units are vector units
- Choose (H)ECC with efficiently vectorizable formulas

Looking back...

- Fastest computation units are vector units
- Choose (H)ECC with efficiently vectorizable formulas
- Formulas "dictate" the scalar multiplication algorithm

Looking back...

- Fastest computation units are vector units
- Choose (H)ECC with efficiently vectorizable formulas
- Formulas “dictate” the scalar multiplication algorithm
- Choose representation of field elements for fast reduction

Looking back...

- Fastest computation units are vector units
- Choose (H)ECC with efficiently vectorizable formulas
- Formulas "dictate" the scalar multiplication algorithm
- Choose representation of field elements for fast reduction
- Adjust formulas according to fast shuffle instructions

Looking back. . .

- Fastest computation units are vector units
- Choose (H)ECC with efficiently vectorizable formulas
- Formulas “dictate” the scalar multiplication algorithm
- Choose representation of field elements for fast reduction
- Adjust formulas according to fast shuffle instructions
- Optimizations go through all levels of the pyramid!

Results

128-bit secure, constant-time scalar multiplication

arch	cycles	open	g	source of software
Sandy	194036	yes	1	Bernstein–Duif–Lange–Schwabe–Yang CHES 2011
Sandy	153000?	no	1	Hamburg
Sandy	137000?	no	1	Longa–Sica Asiacrypt 2012
Sandy	122716	yes	2	Bos–Costello–Hisil–Lauter Eurocrypt 2013
Sandy	119904	yes	1	Oliveira–López–Aranha–Rodríguez-Henríquez CHES 2013
Sandy	96000?	no	1	Faz-Hernández–Longa–Sánchez CT-RSA 2014
Sandy	92000?	no	1	Faz-Hernández–Longa–Sánchez July 2014
Sandy	88916	yes	2	**new (our results)**

Results

128-bit secure, constant-time scalar multiplication

arch	cycles	open	g	source of software
Ivy	182708	yes	1	Bernstein–Duif–Lange–Schwabe–Yang CHES 2011
Ivy	145000?	yes	1	Costello–Hisil–Smith Eurocrypt 2014
Ivy	119032	yes	2	Bos–Costello–Hisil–Lauter Eurocrypt 2013
Ivy	114036	yes	1	Oliveira–López–Aranha–Rodríguez-Henríquez CHES 2013
Ivy	92000?	no	1	Faz-Hernández–Longa–Sánchez CT-RSA 2014
Ivy	89000?	no	1	Faz-Hernández–Longa–Sánchez July 2014
Ivy	88448	yes	2	**new (our results)**

More results

Also optimized for Intel Haswell

arch	cycles	open	g	source of software
Haswell	145907	yes	1	Bernstein–Duif–Lange–Schwabe–Yang CHES 2011
Haswell	100895	yes	2	Bos–Costello–Hisil–Lauter Eurocrypt 2013
Haswell	55595	no	1	Oliveira–López–Aranha–Rodríguez-Henríquez CHES 2013
Haswell	54389	yes	2	**new (our results)**

Even more results

Also optimized for ARM Cortex-A8

arch	cycles	open	g	source of software
A8-slow	497389	yes	1	Bernstein–Schwabe CHES 2012
A8-slow	305395	yes	2	**new (our result)**
A8-fast	460200	yes	1	Bernstein–Schwabe CHES 2012
A8-fast	273349	yes	2	**new (our result)**

Resources online

Paper:
Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, Peter Schwabe. *"Kummer strikes back: new DH speed records"*.
http://cryptojedi.org/papers/#kummer

Software:
Included in SUPERCOP, subdirectory `crypto_scalarmult/kummer/`
http://bench.cr.yp.to/supercop.html