Fast, uniform scalar multiplication for genus 2 Jacobians with fast Kummers

Ping Ngai (Brian) Chung Craig Costello **Benjamin Smith**

University of Chicago

Microsoft Research

INRIA + Laboratoire d'Informatique de l'École polytechnique (LIX)

SAC 2016
St. John's, Canada, 11/08/2016

We want to implement basic cryptosystems
based on the hardness of
the Discrete Logarithm and Diffie–Hellman problems
in some group $\mathcal{G}$.

Especially: Diffie–Hellman Key exchange,
Schnorr and (EC)DSA Signatures, ...

Work to be done

Group operation in $\mathcal{G}$: $\oplus$. Inverse: $\ominus$.

We occasionally need to compute isolated $\oplus$es.

We mostly need to compute *scalar multiplications*:

$$(m, P) \longmapsto [m]P := \underbrace{P \oplus \cdots \oplus P}_{m \text{ times}}$$

for P in $\mathcal{G}$ and m in $\mathbb{Z}$ (with $[-m]P = [m](\ominus P)$).

Side channel safety $\implies$ scalar multiplication must be *uniform* and *constant-time* when the scalar m is secret.

...So you want to instantiate a DLP/DHP-based protocol

Smallest key size for a given security level:
use an *elliptic curve* or a *genus 2 Jacobian*.

For signatures and encryption:

Elliptic: Edwards curves (eg. Ed25519), NIST curves, etc.

Genus 2: Jacobian surfaces.

Scalar mult: *Uniform* genus 2 is much slower than elliptic curves.

For Diffie–Hellman:

Elliptic: x-lines of Montgomery curves (eg. Curve25519)

Genus 2: Kummer surfaces (Jacobians modulo ± 1).

Scalar mult: *Uniform* genus 2 can be faster than elliptic curves.

E.g.: Bos–Costello–Hisil–Lauter (2012)
Bernstein–Chuengsatiansup–Lange–Schwabe (2014)

Our aim: bring Diffie–Hellman performance to signatures in genus 2.

Genus 2 curves

$\mathcal{C} : y^2 = f(x)$ with $f \in \mathbb{F}_p[x]$ degree 5 or 6 and squarefree

Unlike elliptic curves, the points do not form a group.

Making groups from genus 2 curves

Jacobian: algebraic group $\mathcal{J}_\mathcal{C} \sim \mathcal{C}^{(2)}$:
Elements are pairs of points on $\mathcal{C}$,
with all pairs $\{(x, y), (x, -y)\}$ "blown down" to 0.

Negation $\ominus : \{(x_1, y_1), (x_2, y_2)\} \mapsto \{(x_1, -y_1), (x_2, -y_2)\}$

Group law on $\mathcal{J}_\mathcal{C}$ induced by
$\{P_1, P_2\} \oplus \{Q_1, Q_2\} \oplus \{R_1, R_2\} = 0$
whenever $P_1, P_2, Q_1, Q_2, R_1, R_2$ are
the intersection of $\mathcal{C}$ with some cubic $y = g(x)$.

Why? Any 4 points in the plane determine a cubic $y = g(x)$, which must intersect $\mathcal{C} : y^2 = f(x)$ in 6 points because $g(x)^2 = f(x)$ has 6 solutions.

Genus 2 group law: $\{P_1, P_2\} \oplus \{Q_1, Q_2\} = \ominus\{R_1, R_2\} = \{S_1, S_2\}$

Algorithmically: we use the Mumford representation and Cantor's algorithm.

Why is uniform genus 2 tricky?

Elements $\{P_1, P_2\}$: separate, *incompatible* representations for cases where one or both of the P_i are at infinity.

Group law $\{P_1, P_2\} \oplus \{Q_1, Q_2\} = \{S_1, S_2\}$: branch-tacular, separate special cases for P_i, Q_i at infinity, for $P_i = P_j$, for $P_i = Q_j$, for $\{P_1, P_2\} = \{Q_1, Q_2\}$, ...

These special cases are never implemented in "record-breaking" genus 2 implementations, but they're easy to attack in practice.

For elliptic curves, we can always sweep the special cases under a convenient line to get a uniform group law, but in genus 2 this is much harder; *protection kills performance*.

Why is Diffie–Hellman different?

Now you know why genus 2 Jacobians are painful candidates for cryptographic groups.

So why is genus 2 fast and safe for Diffie–Hellman?

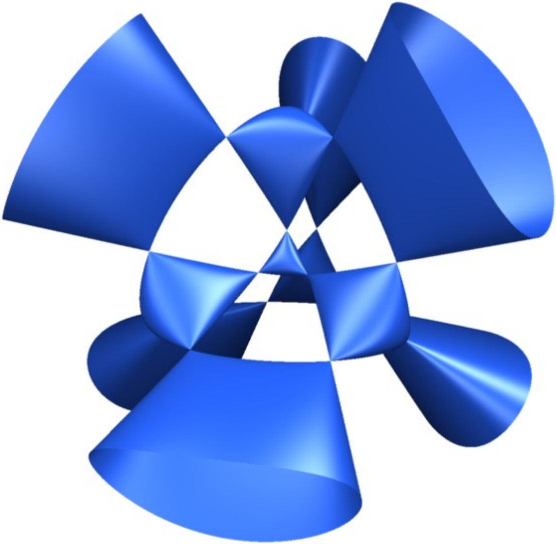
Because DH *doesn't need a group law*, just scalar multiplication.

So we can "drop signs" and work modulo $\ominus$, on the Kummer surface

$$\mathcal{K}_{\mathcal{C}} := \mathcal{J}_{\mathcal{C}}/\langle\pm 1\rangle \ .$$

Elliptic curve equivalent: work on x-line $\mathbb{P}^1$*, eg. Curve25519 (Bernstein 2006).*

What a Kummer surface looks like



Moving from $\mathcal{J}_\mathcal{C}$ to the Kummer $\mathcal{K}_\mathcal{C}$

Quotient map $x : \mathcal{J}_\mathcal{C} \longrightarrow \mathcal{K}_\mathcal{C}$ (ie $x(P) = \pm P$)

No group law on $\mathcal{K}_\mathcal{C}$: $x(P)$ and $x(Q)$ determines $x(P \oplus Q)$ and $x(P \ominus Q)$, but we can't tell which is which.

Still, $\ominus[m](P) = [m](\ominus P)$ for any $m \in \mathbb{Z}$ and $P \in \mathcal{J}_\mathcal{C}$, so we do have a "scalar multiplication" on $\mathcal{K}_\mathcal{C}$:

$$[m] : x(P) \longmapsto x([m]P) \ .$$

Problem: How do we compute $[m]$ efficiently, *without* $\oplus$?

Any 3 of $x(P)$, $x(Q)$, $x(P \ominus Q)$, and $x(P \oplus Q)$ determines the 4th, so we can define

pseudo-addition

$$\mathtt{xADD} : (x(P), x(Q), x(P \ominus Q)) \longmapsto x(P \oplus Q)$$

pseudo-doubling

$$\mathtt{xDBL} : x(P) \longmapsto x([2]P)$$

Bonus: easier to hide/avoid special cases in `xADD` than $\oplus$.

$\Longrightarrow$ Evaluate $[m]$ by combining `xADD`s and `xDBL`s using differential addition chains

(ie. every $\oplus$ has summands with known difference).

Classic example: the Montgomery ladder.

Algorithm 1 The Montgomery ladder

function LADDER($m = \sum_{i=0}^{\beta-1} m_i 2^i,\ P$)
  $(R_0, R_1) \leftarrow (0, P)$
  for $i := \beta - 1$ down to 0 **do**
    $(R_{m_i}, R_{\neg m_i}) \leftarrow ([2]R_{m_i}, R_{m_i} \oplus R_{\neg m_i})$
  end for $\triangleright$ invariant: $(R_0, R_1) = ([\lfloor m/2^i \rfloor]P, [\lfloor m/2^i \rfloor + 1]P)$
  return R_0 $\triangleright$ $R_0 = [m]P,\ R_1 = [m]P \oplus P$
end function

For each group operation $R_0 \oplus R_1$, the difference $R_0 \ominus R_1$ is *fixed*
$\implies$ trivial adaptation from $\mathcal{J}_{\mathcal{C}}$ to $\mathcal{K}_{\mathcal{C}}$

Algorithm 2 The Montgomery ladder on the Kummer

function $\textsc{Ladder}(m = \sum_{i=0}^{\beta-1} m_i 2^i, \pm P)$
    $(x_0, x_1) \leftarrow (x(0), x(P))$
    for $i := \beta - 1$ down to 0 **do**
        $(x_{m_i}, x_{\neg m_i}) \leftarrow (\mathtt{xDBL}(x_{m_i}), \mathtt{xADD}(x_0, x_1, x(P)))$
    end for $\triangleright$ invariant: $x_0 = x([\lfloor m/2^i \rfloor]P$, $x_1 = x([\lfloor m/2^i \rfloor + 1]P)$
    return x_0 $(= x([m]P))$
end function

High symmetry of $\mathcal{K}_{\mathcal{C}}$ $\implies$ fast, vectorizable `xADD` and `xDBL` (Gaudry)
$\implies$ very fast Kummer-based Diffie–Hellman implementations
Eg. Bos–Costello–Hisil–Lauter (2013),
Bernstein–Chuengsatiansup–Lange–Schwabe (2014).

Pulling a y-rabbit out of an x-hat

Kummer multiplication computes $x([m]P)$ from $x(P)$
—but we need $[m]P$ for signatures...

Mathematically, we threw away the sign:
you can't deduce $[m]P$ from P and $x([m]P)$.

But there's a trick: if you computed $x([m]P)$
using the Montgomery ladder, then you can!

At the end of the loop, $x_0 = x([m]P)$ and $x_1 = x([m]P \oplus P)$;
and P, $x(Q)$, and $x(Q \oplus P)$ uniquely determines Q (for any Q).

Our paper: efficiently computing this in genus 2, with
1D (Montgomery) and 2D (Bernstein) SM algorithms.

P, $x(Q)$, *and* $x(P \oplus Q)$ *determine* Q

This is an old trick for elliptic curves: cf. López–Dahab (CHES 99), Okeya–Sakurai (CHES 01), Brier–Joye (PKC 02).

Genus 2 group law: $\{P_1, P_2\} \oplus \{Q_1, Q_2\} = \{S_1, S_2\}$

Choosing $\{T_1, T_2\}$ as (the wrong) preimage of $x(\{Q_1, Q_2\})$ yields a cubic incompatible with $x(\{S_1, S_2\})$.

So: your fast Kummer implementations can now be easily upgraded to full Jacobian group implementations.

Fast Diffie–Hellman code now yields efficient signatures.

Algorithm 3 Montgomery/Kummer-based multiplication on the Jacobian

function $\textsc{ScalarMultiply}(m = \sum_{i=0}^{\beta-1} m_i 2^i,\ P)$
    $(x_0, x_1) \leftarrow (x(0), x(P))$
    for $i := \beta - 1$ down to 0 **do** ▷ Montgomery ladder
        $(x_{m_i}, x_{\neg m_i}) \leftarrow (\mathtt{xDBL}(x_{m_i}), \mathtt{xADD}(x_0, x_1, x(P)))$
    end for ▷ invariant: $x_0 = x([\lfloor m/2^i \rfloor]P)$, $x_1 = x([\lfloor m/2^i \rfloor + 1]P)$
    $Q \leftarrow \mathtt{Recover}(P, x_0, x_1)$ ▷ $Q = [m]P$
    return Q
end function

Gratuitous cross-promotion

...this isn't just wishful theory.

Our technique was used in μKummer: efficient Diffie–Hellman *and* Schnorr signatures for microcontrollers (Renes–Schwabe–S.–Batina, CHES 2016)

Comparison for 8-bit architecture (AVR ATmega):

Protocol	Object	kCycles	Stack bytes
Diffie–Hellman	Curve25519	13900	494
	μ**Kummer**	9513 (68%)	99 (20%)
Schnorr signing	Ed25519	19048	1473
	μ**Kummer**	10404 (55%)	926 (63%)
Schnorr verifying	Ed25519	30777	1226
	μ**Kummer**	16241 (53%)	992 (75%)

(vs. Curve25519: Düll-Haase-Hinterwälder-Hutter-Paar-Sánchez-Schwabe, Ed25519: Nascimento-López-Dahab)

Comparison for 32-bit architecture (ARM Cortex M0):

Multiplication for	Object	kCycles	Stack bytes
Diffie–Hellman	Curve25519	3590	548
	μ**Kummer**	2634 (73%)	248 (45%)
Schnorr	NIST-P256	10730	540
	μ**Kummer**	2709 (25%)	968 (179%)

(vs. Curve25519: Düll-Haase-Hinterwälder-Hutter-Paar-Sánchez-Schwabe, NIST-P256: Wenger-Unterluggauer-Werner)