

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

Ping Ngai Chung, Craig Costello, Benjamin Smith

▶ **To cite this version:**

Ping Ngai Chung, Craig Costello, Benjamin Smith. Fast, uniform scalar multiplication for genus 2 Jacobians with fast Kummers. Selected Areas in Cryptography - SAC 2016, Aug 2016, St John's, Canada. pp.18, 10.1007/978-3-319-69453-5_25 . hal-01353480

HAL Id: hal-01353480

https://inria.hal.science/hal-01353480

Submitted on 10 Nov 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

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

Ping Ngai Chung[1], Craig Costello[2], and Benjamin Smith[3]

[1] University of Chicago, USA
briancpn@math.uchicago.edu
[2] Microsoft Research, USA
craigco@microsoft.com
[3] INRIA *and* Laboratoire d'Informatique de l'École polytechnique (LIX), Palaiseau, France
smith@lix.polytechnique.fr

Abstract. We give one- and two-dimensional scalar multiplication algorithms for Jacobians of genus 2 curves that operate by projecting to Kummer surfaces, where we can exploit faster and more uniform pseudo-multiplication, before recovering the proper "signed" output back on the Jacobian. This extends the work of López and Dahab, Okeya and Sakurai, and Brier and Joye to genus 2, and also to two-dimensional scalar multiplication. The technique is especially interesting in genus 2, because Kummer surfaces can outperform comparable elliptic curve systems.

Keywords: Kummer surface, genus 2, scalar multiplication, signatures, pseudomultiplication, uniform, constant-time.

1 Introduction

In this article we show how to exploit Gaudry's fast, uniform Kummer surface arithmetic [15] to carry out full scalar multiplications on genus 2 Jacobians. This brings the speed and side-channel security of Kummers, so far only used for Diffie–Hellman implementations, to implementations of other discrete-log-based cryptographic protocols including signature schemes.[4]

To make things precise, let $\mathcal{J}_\mathcal{C}$ be the Jacobian of a genus 2 curve $\mathcal{C}$ over a finite field $\mathbb{F}_q$ of characteristic > 3 (with $\oplus$ denoting the group law on $\mathcal{J}_\mathcal{C}$, and $\ominus$ the inverse). We want to compute scalar multiplications

$$(m, P) \longmapsto [m]P := \underbrace{P \oplus \cdots \oplus P}_{m \text{ times}} \quad \text{for } m \in \mathbb{Z}_{\geq 0} \text{ and } P \in \mathcal{J}_\mathcal{C}(\mathbb{F}_q) \ ,$$

which are at the heart of all discrete logarithm and Diffie–Hellman problem-based cryptosystems. If the scalar m is secret, then $[m]P$ must be computed in

[4] This article supersedes the much longer unpublished manuscript [9], which can be found at http://eprint.iacr.org/2015/983. The longer version includes algorithms for scalar multiplication for general genus 2 Jacobians that are *not* equipped with fast Kummer surfaces, and proposes a signature scheme based on these results.

a *uniform* and *constant-time* way to protect against even the most elementary side-channel attacks. This means that the execution path of the algorithm must be independent of the scalar m (we may assume that the bitlength of m is fixed).

The quotient *Kummer surface* $\mathcal{K}_\mathcal{C} := \mathcal{J}_\mathcal{C}/\langle \pm 1 \rangle$ identifies group elements with their inverses (this is the genus-2 analogue of projecting elliptic curve points onto the x-coordinate). If P is a point on $\mathcal{J}_\mathcal{C}$, then $\pm P$ denotes its image in $\mathcal{K}_\mathcal{C}$. Scalar multiplication on $\mathcal{J}_\mathcal{C}$ induces a well-defined *pseudomultiplication*

$$(m, \pm P) \longmapsto \pm[m]P \quad \text{for } m \in \mathbb{Z}_{\geq 0} \text{ and } P \in \mathcal{J}_\mathcal{C}(\mathbb{F}_q) \ ,$$

which can be computed using differential addition chains in the exact analogue of x-only arithmetic for elliptic curves. This suffices for implementing protocols like Diffie–Hellman key exchange which *only* involve scalar multiplication, as Bernstein's Curve25519 software did for elliptic curves [1]. But we emphasize that $\mathcal{K}_\mathcal{C}$ is *not* a group, and its lack of a group operation prevents us instantiating many group-based protocols in $\mathcal{K}_\mathcal{C}$ (see [26, §5]).

It has long been known that x-only pseudomultiplication can be used for full scalar multiplication on elliptic curves: López and Dahab [20] (followed by Okeya and Sakurai [23] and Brier and Joye [4]) showed that the auxiliary values computed by the x-only Montgomery ladder can be used to recover the missing y-coordinate, and hence to compute full scalar multiplications on elliptic curves. The main innovation of this paper is to extend this technique from elliptic curves to genus 2, and from one- to two-dimensional scalar multiplication. This allows cryptographic protocols instantiated in genus-2 Jacobians to delegate their scalar multiplications to faster, more uniform Kummer surfaces.

In the abstract, our algorithms follow the same common pattern:

1. `Project` the inputs from $\mathcal{J}_\mathcal{C}$ to $\mathcal{K}_\mathcal{C}$;
2. Pseudomultiply in $\mathcal{K}_\mathcal{C}$ using a differential addition chain, such as the Montgomery ladder [22] or Bernstein's binary chain [2];
3. `Recover` the correct preimage for the full scalar multiplication in $\mathcal{J}_\mathcal{C}$ from the outputs of the pseudomultiplication, using our new Algorithm 2.

More concretely, if $\mathcal{J}_\mathcal{C}$ is a genus-2 Jacobian admitting a fast Kummer surface as in §2, and $\mathcal{B} \subset \mathcal{J}_\mathcal{C}(\mathbb{F}_q)$ is the set of Definition 1, then our main results are

Theorem 1 (`Project` + Montgomery ladder + `Recover`): If P is a point in $\mathcal{J}_\mathcal{C}(\mathbb{F}_q) \setminus \mathcal{B}$ then for any β-bit integer m, Algorithm 3 computes $[m]P$ in $(7\beta + 115)\mathtt{M} + (12\beta + 8)\mathtt{S} + (12\beta + 4)\mathtt{m_c} + (32\beta + 79)\mathtt{a} + 2\mathtt{I}$.

Theorem 2 (`Project` + Bernstein's binary chain + `Recover`): If P and Q are points in $\mathcal{J}_\mathcal{C}(\mathbb{F}_q) \setminus \mathcal{B}$ with $P \oplus Q$ and $P \ominus Q$ not in $\mathcal{B}$ and m and n are positive β-bit integers, then Algorithm 4 computes $[m]P \oplus [n]Q$ in $(14\beta + 203)\mathtt{M} + (20\beta + 16)\mathtt{S} + (16\beta + 16)\mathtt{m_c} + (56\beta + 138)\mathtt{a} + 3\mathtt{I}$.

Both algorithms are uniform with respect to their scalars. The two-dimensional multiscalar multiplications of Theorem 2 appear explicitly in many cryptographic protocols (such as Schnorr signature verification), but they are also a

key ingredient in endomorphism-accelerated one-dimensional scalar multiplication techniques like GLV [14] and its descendants.[5]

There are two key benefits to this approach: speed and uniformity. For speed, we note that Gaudry's Kummer arithmetic is markedly faster than full Jacobian arithmetic, and competitive Diffie–Hellman implementations have shown that Kummer-based scalar multiplication software can outperform its elliptic equivalent [3]. Our results bring this speed to a wider range of protocols, such as ElGamal and signature schemes. Indeed, the methods described below (including Algorithms 2 and 3) have already been successfully put into practice in a fast and compact implementation of Schnorr signatures for microcontrollers [24], but without any proof of correctness or explanation of the algorithms[6]; this article provides that proof, and detailed algorithms to enable further implementations.

The second benefit is side-channel protection. Fast, uniform, constant-time algorithms for elliptic curve scalar multiplication are well-known and widely-used. In contrast, for genus 2 Jacobians, the uniform and constant-time requirements are problematic: conventional Cantor arithmetic [6] and its derivatives [17] are highly susceptible to simple side-channel attacks. The explicit formulæ derived for generic additions in Jacobians fail to compute correct results when one or both of the inputs are so-called "special" points (essentially, those corresponding to degree-one divisors on $\mathcal{C}$). While special points are rare enough that random scalar multiplications never encounter them, they are plentiful enough that attackers can easily mount exceptional procedure attacks [18], forcing software into special cases and using timing variations to recover secret data. It has appeared impossible to implement traditional genus 2 arithmetic in a uniform way without abandoning all hope of competitive efficiency [11]. The Jacobian point recovery method we present in §3 solves the problem of uniform genus 2 arithmetic (at least for scalar multiplication): rather than wrestling with the special cases of Cantor's algorithm on $\mathcal{J}_\mathcal{C}$, we can pseudomultiply on the Kummer and then recover the correct image on $\mathcal{J}_\mathcal{C}$.

Remark 1. Robert and Lubicz [21] use similar techniques to speed up their arithmetic for general abelian varieties based on theta functions, viewing the results of the Montgomery ladder on a g-dimensional Kummer variety K as a point on the corresponding abelian variety A embedded in K^2. In contrast to our method, Robert and Lubicz cannot treat A as a Jacobian (since general abelian varieties of dimension $g > 3$ are not Jacobians); so in the case of genus $g = 2$, there is no explicit connection with any curve $\mathcal{C}$, and the starting and finishing points do not involve the Mumford representation. Kohel [19] explores similar ideas for elliptic curves, leading to an interesting interpretation of Edwards curve arithmetic.

Remark 2. Since our focus here is on fast cryptographic implementations, for lack of space, in this article we restrict our attention to curves and Jacobians whose Kummer surfaces have so-called "fast" models (see §2). This implies that

[5] Our techniques should readily extend to the higher-dimensional differential addition chains described by Brown [5]. We do not investigate this here.

[6] The implementation in [24] was based on our longer manuscript [9].

all of our Jacobians have full rational 2-torsion. Our techniques generalize without any difficulty to more general curves and Kummer surfaces, and then replacing the fast Kummer operations described in Appendix A with more general methods wherever they appear in Algorithms 3 and 4 yields efficient, uniform scalar multiplication algorithms for any genus 2 Jacobian.

Notation As usual, $\mathtt{M}$, $\mathtt{S}$, $\mathtt{I}$, and $\mathtt{a}$ denote the costs of one multiplication, squaring, inversion, and addition in $\mathbb{F}_q$, respectively; for simplicity, we assume subtraction and unary negation in $\mathbb{F}_q$ also cost $\mathtt{a}$. We let $\mathtt{m_c}$ denote the cost of multiplication by the theta constants a, b, c, d, A, B, C, D of §2 and their inverses (we aim to make these as small as possible). We assume we have efficient constant-time conditional selection and swap routines: $\mathtt{SELECT}(b, (X_0, X_1))$ returns X_b, and $\mathtt{SWAP}(b, (X_0, X_1))$ returns (X_b, X_{1-b}) (see Appendix B for sample code).

2 Genus 2 Jacobians with fast Kummer Surfaces

Suppose we have a, b, c, and d in $\mathbb{F}_q \setminus \{0\}$ such that if we set

$$A := a+b+c+d\ , \qquad B := a+b-c-d\ ,$$
$$C := a-b+c-d\ , \qquad D := a-b-c+d\ ,$$

then $abcdABCD \neq 0$ and $CD/(AB) = \alpha^2$ for some α in $\mathbb{F}_q$. Setting

$$\lambda := a/b \cdot c/d\ , \quad \mu := c/d \cdot (1+\alpha)/(1-\alpha)\ , \quad \nu := a/b \cdot (1+\alpha)/(1-\alpha)\ ,$$

we define an associated genus 2 curve $\mathcal{C}$ in Rosenhain form:

$$\mathcal{C} : y^2 = f(x) = x(x-1)(x-\lambda)(x-\mu)(x-\nu)\ ,$$

so $f(x) = x^5 + f_4x^4 + f_3x^3 + f_2x^2 + f_1x$ with $f_4 = -(\lambda+\mu+\nu+1)$, $f_3 = \lambda\mu + \lambda\nu + \lambda + \mu\nu + \mu + \nu$, $f_2 = -(\lambda\mu\nu + \lambda\mu + \lambda\nu + \mu\nu)$, $f_1 = \lambda\mu\nu$. (The techniques in this paper extend to far more general genus 2 curves—see the manuscript [9]—but every fast Kummer is associated with a curve in this form.)

Elements of $\mathcal{J}_\mathcal{C}(\mathbb{F}_q)$ are presented in their standard Mumford representation:

$$P \in \mathcal{J}_\mathcal{C}(\mathbb{F}_q) \longleftrightarrow \langle a(x) = x^2 + a_1x + a_0, b(x) = b_1x + b_0\rangle$$

where a_1, a_0, b_1, and b_0 are in $\mathbb{F}_q$ and $b(x)^2 \equiv f(x) \pmod{a(x)}$. The group law on $\mathcal{J}_\mathcal{C}$ is typically computed using Cantor's algorithm, specialized to genus 2. Here we suppose we have a function $\mathtt{JacADD} : (P, Q) \mapsto P \oplus Q$ which computes the group law as in [17, Eq. (12)] at a cost of $22\mathtt{M} + 2\mathtt{S} + 1\mathtt{I} + 27\mathtt{a}$.

The *fast Kummer surface* for $\mathcal{C}$ is the quartic surface $\mathcal{K}_\mathcal{C}^{\text{fast}} \subset \mathbb{P}^3$ defined by

$$\mathcal{K}_\mathcal{C}^{\text{fast}} : \begin{pmatrix} (X^2+Y^2+Z^2+T^2) \\ -F(XT+YZ) - G(XZ+YT) - H(XY+ZT) \end{pmatrix}^2 = EXYZT \quad (1)$$

where

$$F = \frac{a^2 - b^2 - c^2 + d^2}{ad - bc}\ , \quad G = \frac{a^2 - b^2 + c^2 - d^2}{ac - bd}\ , \quad H = \frac{a^2 + b^2 - c^2 - d^2}{ab - cd}\ ,$$

and $E = 4abcd\,(ABCD/((ad-bc)(ac-bd)(ab-cd)))^2$. These surfaces were algorithmically developed by the Chudnovskys [8], and introduced in cryptography by Gaudry [15]; here we use the "squared-theta" model of [10, Ch. 4]. Cryptographic parameters for genus-2 Jacobians equipped with fast Kummers can be (and have been) computed: the implementation of [24] uses the parameters from [16] in the algorithms presented below.

The map $\texttt{Project} : \mathcal{J}_\mathcal{C} \to \mathcal{K}_\mathcal{C}^{\text{fast}}$ mapping P to $\pm P$ is classical (cf. [10, §5.3]), and implemented by Algorithm 1. It is not uniform or constant-time, but it does not need to be: in most applications the input points are already public.

Algorithm 1: $\texttt{Project}$: $\mathcal{J}_\mathcal{C} \to \mathcal{K}_\mathcal{C}^{\text{fast}}$.

Input: $P \in \mathcal{J}_\mathcal{C}(\mathbb{F}_q)$
Output: $\pm P \in \mathcal{K}_\mathcal{C}^{\text{fast}}(\mathbb{F}_q)$
Cost: $8\mathtt{M} + 1\mathtt{S} + 4\mathtt{m_c} + 14\mathtt{a}$, assuming precomputed $\lambda\mu$, $\lambda\nu$.

if $P = 0$ **then return** $(a : b : c : d)$
else if $P = \langle x - u, v\rangle$ **then**
  $(\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3, \mathtt{t}_4) \leftarrow (u - 1, u - \lambda, u - \mu, u - \nu)$ // 4a
  return $(a \cdot \mathtt{t}_1 \cdot \mathtt{t}_3 : b \cdot \mathtt{t}_2 \cdot \mathtt{t}_4 : c \cdot \mathtt{t}_1 \cdot \mathtt{t}_4 : d \cdot \mathtt{t}_2 \cdot \mathtt{t}_3)$ // 4M+4$\mathtt{m_c}$
else (generic case $P = \langle x^2 + a_1 x + a_0, b_1 x + b_0\rangle$)
  $(\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3) \leftarrow (a_1 + \lambda, a_1 + 1, b_0^2)$ // 1S+2a
  $(\mathtt{t}_4, \mathtt{t}_5) \leftarrow (a_0 \cdot (a_0 - \mu) \cdot (\mathtt{t}_1 + \nu), a_0 \cdot (a_0 - \lambda\nu) \cdot (\mathtt{t}_2 + \mu))$ // 4M+4a
  $(\mathtt{t}_6, \mathtt{t}_7) \leftarrow (a_0 \cdot (a_0 - \nu) \cdot (\mathtt{t}_1 + \mu), a_0 \cdot (a_0 - \lambda\mu) \cdot (\mathtt{t}_2 + \nu))$ // 4M+4a
  return $(a \cdot \mathtt{t}_4 + \mathtt{t}_3, b \cdot \mathtt{t}_5 + \mathtt{t}_3, c \cdot \mathtt{t}_6 + \mathtt{t}_3, d \cdot \mathtt{t}_7 + \mathtt{t}_3)$ // 4$\mathtt{m_c}$+4a

Table 1 summarizes the key standard operations on $\mathcal{K}_\mathcal{C}^{\text{fast}}$ and their costs (for detailed pseudocode, see Appendix A). The pseudo-doubling $\texttt{xDBL}$ is correct on all inputs; the pseudo-additions $\texttt{xADD}^*$, $\texttt{xADD}$ and combined pseudo-double-and-add $\texttt{xDBLADD}$ are correct for all inputs provided the difference point has no coordinate equal to zero. Since almost all difference points are fixed in our algorithms, and these "bad" points are extremely rare (there are only $O(q)$ of them, versus $O(q^2)$ other points), we simply prohibit them as input: Definition 1 identifies their preimages in $\mathcal{J}_\mathcal{C}$ for easy identification and rejection.

Definition 1. *Let $\mathcal{B} \subset \mathcal{J}_\mathcal{C}(\mathbb{F}_q)$ be the set of elements P whose images $\pm P$ in $\mathcal{K}_\mathcal{C}^{\text{fast}}$ have a zero coordinate; or equivalently, $P = \langle x^2 + a_1 x + a_0, b_1 x + b_0\rangle$ with*

1. $(\mu a_1 + a_0)(1(a_1 + \lambda + \nu) + a_0) + (\lambda\mu - \lambda\nu + \mu\nu - 1\mu)a_0 + \lambda\mu\nu = 0$, *or*
2. $(\nu a_1 + a_0)(\lambda(a_1 + 1 + \mu) + a_0) - (\lambda\nu - \mu\nu + 1\mu - 1\nu)a_0 + \lambda\mu\nu = 0$, *or*
3. $(\nu a_1 + a_0)(1(a_1 + \lambda + \mu) + a_0) - (\lambda\mu - \lambda\nu - \mu\nu + 1\nu)a_0 + \lambda\mu\nu = 0$, *or*
4. $(\mu a_1 + a_0)(\lambda(a_1 + 1 + \nu) + a_0) - (\lambda\mu - \mu\nu - 1\mu + 1\nu)a_0 + \lambda\mu\nu = 0$.

Table 1. Operations on $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$ and $\mathcal{J}_{\mathcal{C}}$. All but `JacADD` are uniform. The operations `xADD`*, `xADD`, and `xDBLADD` require $P \ominus Q \notin \mathcal{B}$.

Algorithm	Operation: Input $\mapsto$ Output	**M**	**S**	$\mathbf{m}_c$	**a**	**I**
`JacADD`	$(P, Q) \mapsto P \oplus Q$	22	2	0	27	1
`xDBL`	$\pm P \mapsto \pm[2]P$	0	8	8	16	0
`xADD`*	$(\pm P, \pm Q, \pm(P \ominus Q)) \mapsto \pm(P \oplus Q)$	14	4	4	24	0
`xADD`	$(\pm P, \pm Q, \texttt{Wrap}(\pm(P \ominus Q))) \mapsto \pm(P \oplus Q)$	7	4	4	24	0
`xDBLADD`	$(\pm P, \pm Q, \texttt{Wrap}(\pm(P \ominus Q))) \mapsto (\pm[2]P, \pm(P \oplus Q))$	7	12	12	32	0
`Wrap`	$(x : y : z : t) \mapsto (x/y, x/z, x/t)$	7	0	0	0	1
`Wrap4`	$(\pm P_i)_{i=1}^4 \mapsto (\texttt{Wrap}(\pm P_i))_{i=1}^4$	37	0	0	0	1

To optimize pseudo-additions, we define a mapping $\texttt{Wrap} : (x : y : z : t) \mapsto (x/y, x/z, x/t)$ (for $(x : y : z : y)$ not in $\mathcal{B}$). To `Wrap` one Kummer point costs $7\mathtt{M} + 1\mathtt{I}$, but saves $7\mathtt{M}$ in every subsequent pseudo-addition with that point as its difference. In Algorithm 4 we need to `Wrap` four points; `Wrap4` does this with a single shared inversion, for a total cost of $37\mathtt{M} + 1\mathtt{I}$.

3 Point recovery in genus 2

Our aim is to compute scalar multiplications $(m, P) \mapsto R = [m]P$ on $\mathcal{J}_{\mathcal{C}}$. `Projecting` to $\mathcal{K}_{\mathcal{C}}$ yields $\pm P$, and then pseudomultiplication (which we will describe below) gives $\pm R = \pm[m]P$; but it can also produce $\pm(R \oplus P)$ as an auxiliary output. We will reconstruct R from this data, by defining a map

$$\texttt{Recover} : (P, \pm P, \pm R, \pm(R \oplus P)) \longmapsto R \quad \text{for } P \text{ and } R \in \mathcal{J}_{\mathcal{C}} \ .$$

The map $\mathcal{J}_{\mathcal{C}} \to \mathcal{K}_{\mathcal{C}}^{\text{fast}}$ factors through the "general Kummer" $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$, another quartic surface in $\mathbb{P}^3$ defined (as in [7, Ch. 3], taking $f_6 = f_0 = 0$ and $f_5 = 1$, and using coordinates ξ_i, to avoid confusion with $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$) by

$$\mathcal{K}_{\mathcal{C}}^{\text{gen}} : K_2(\xi_1, \xi_2, \xi_3)\xi_4^2 + K_1(\xi_1, \xi_2, \xi_3)\xi_4 + K_0(\xi_1, \xi_2, \xi_3) = 0 \tag{2}$$

where $K_2 = \xi_2^2 - 4\xi_1\xi_3$, $K_1 = -2(f_1\xi_1^2 + f_3\xi_1\xi_3 + f_5\xi_3^2)\xi_2 - 4\xi_1\xi_3(f_2\xi_1 + f_4\xi_3)$, and $K_0 = (f_1\xi_1^2 - f_3\xi_1\xi_3 + f_5\xi_3^2)^2 - 4\xi_1\xi_3(f_1\xi_2 + f_2\xi_3)(f_4\xi_1 + f_5\xi_2)$. While fast Kummers offer significant gains in performance and uniformity, this comes at the price of full rational 2-torsion: hence, not every Kummer can be put in fast form. But the general Kummer exists for all genus 2 curves, not just those admitting a fast Kummer; roughly speaking, $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$ is the analogue of the x-line of the Weierstrass model of an elliptic curve, while $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$ corresponds to the x-line of a Montgomery model.[7] As such, $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$ is much more naturally related to the

[7] The use of $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$ in cryptography was investigated by Smart and Siksek [26] and Duquesne [13]. The polynomials defining pseudo-operations on $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$ (see [7, §3.4]) are hard to evaluate quickly, and do not offer competitive performance. However, they are completely compatible with our `Project`-pseudomultiply-`Recover` pattern, and we could use them to construct uniform and constant-time scalar multiplication algorithms for genus 2 Jacobians that do *not* admit fast Kummers.

Mumford model of $\mathcal{J}_\mathcal{C}$; so it makes sense to map our recovery problem from $\mathcal{K}_\mathcal{C}^{\text{fast}}$ into $\mathcal{K}_\mathcal{C}^{\text{gen}}$ and then recover from $\mathcal{K}_\mathcal{C}^{\text{gen}}$ to $\mathcal{J}_\mathcal{C}$.

The map $\pi : \mathcal{J}_\mathcal{C} \to \mathcal{K}_\mathcal{C}^{\text{gen}}$ is described in [7, Eqs. (3.1.3–5)]; it maps generic points $\langle x^2 + a_1 x + a_0, b_1 x + b_0 \rangle$ in $\mathcal{J}_\mathcal{C}$ to $(\xi_1 \colon \xi_2 \colon \xi_3 \colon \xi_4)$ in $\mathcal{K}_\mathcal{C}^{\text{gen}}$, where

$$(\xi_1 : \xi_2 : \xi_3 : \xi_4) = (1 : -a_1 : a_0 : b_1^2 + (a_1^2 - a_0)a_1 + a_1(f_3 - f_4 a_1) - f_2) \ . \quad (3)$$

Projecting onto the $(\xi_1 : \xi_2 : \xi_3)$-plane yields a natural double cover $\rho : \mathcal{K}_\mathcal{C}^{\text{gen}} \to \mathbb{P}^2$; comparing with (3), we see that $\rho \circ \pi$ corresponds to projecting onto the a-polynomial of the Mumford representation.

Proposition 1. *Suppose $P = \langle x^2 + a_1^P x + a_0^P, b_1^P x + b_0^P \rangle$ and $R = \langle x^2 + a_1^R x + a_0^R, b_1^R x + b_0^R \rangle$ are in $\mathcal{J}_\mathcal{C}(\mathbb{F}_q)$. Let $(\xi_1^R : \xi_2^R : \xi_3^R : \xi_4^R) = \pi(R)$ in $\mathcal{K}_\mathcal{C}^{\text{gen}}$, and let $(\xi_1^\oplus : \xi_2^\oplus : \xi_3^\oplus) = \rho(\pi(P \oplus R))$ and $(\xi_1^\ominus : \xi_2^\ominus : \xi_3^\ominus) = \rho(\pi(P \ominus R))$ in $\mathbb{P}^2$. Let $Z_1 = \xi_2^R + a_1^P \xi_1^R$, $Z_2 = \xi_3^R - a_0^P \xi_1^R$, and $Z_3 = -(a_1^P \xi_3^R + a_0^P \xi_2^R)$. Then*

$$(\xi_1^R)^2 (b_1^R, b_0^R) = (G_3, G_4) \begin{pmatrix} \xi_2^R Z_1 - \xi_1^R Z_2 & -\xi_1^R Z_1 \\ -\xi_3^R Z_1 & \xi_1^R Z_2 \end{pmatrix} \quad (4)$$

where G_3 and G_4 satisfy

$$C(G_3, G_4) = D(G_1, G_2) \begin{pmatrix} \xi_1^\oplus \xi_3^\ominus - \xi_3^\oplus \xi_1^\ominus & \xi_2^\oplus \xi_3^\ominus - \xi_3^\oplus \xi_2^\ominus \\ \xi_2^\oplus \xi_1^\ominus - \xi_1^\oplus \xi_2^\ominus & \xi_3^\oplus \xi_1^\ominus - \xi_1^\oplus \xi_3^\ominus \end{pmatrix} \quad (5)$$

where $\xi_1^R D = Z_2^2 - Z_1 Z_3$ and G_1 and G_2 satisfy

$$D(G_1, G_2) = (b_1^P, b_0^P) \begin{pmatrix} Z_2 & a_0^P Z_1 \\ Z_1 & -a_1^P Z_1 - Z_2 \end{pmatrix} \quad (6)$$

and

$$C = \frac{-2D \left(2\xi_1^\oplus \xi_1^\ominus G_2^2 - (\xi_2^\oplus \xi_1^\ominus + \xi_1^\oplus \xi_2^\ominus) G_1 G_2 + (\xi_3^\oplus \xi_1^\ominus + \xi_1^\oplus \xi_3^\ominus) G_1^2 \right)}{G_1^2 + G_3^2} \ .$$

Proof. This is a disguised form of the geometric group law on $\mathcal{J}_\mathcal{C}$ (cf. [7, §1.2]). The points P and R correspond to unique degree-2 divisor classes on $\mathcal{C}$: say,

$$P \longleftrightarrow [(u_P, v_P) + (u'_P, v'_P)] \quad \text{and} \quad R \longleftrightarrow [(u_R, v_R) + (u'_R, v'_R)] \ .$$

(We do not compute the values of $u_P, v_P, u'_P, v'_P, u_R, v_R, u'_R$, and v'_R, which are generally in $\mathbb{F}_{q^2}$; they are purely formal devices here.) Let

$$E_1 = \frac{v_P}{(u_P - u'_P)(u_P - u_R)(u_P - u'_R)} \ , \quad E_2 = \frac{v'_P}{(u'_P - u_P)(u'_P - u_R)(u'_P - u'_R)} \ ,$$
$$E_3 = \frac{v_R}{(u_R - u'_P)(u_R - u_P)(u_R - u'_R)} \ , \quad E_4 = \frac{v'_R}{(u'_R - u_P)(u'_R - u'_P)(u'_R - u_R)} \ .$$

The functions $G_1 := E_1 + E_2$, $G_2 := u'_P E_1 + u_P E_2$, $G_3 := E_3 + E_4$, and $G_4 := u'_R E_3 + u_R E_4$ are functions of P and R, because they are symmetric with

respect to $(u_P, v_P) \leftrightarrow (u'_P, v'_P)$ and $(u_R, v_R) \leftrightarrow (u'_R, v'_R)$. Now, the geometric expression of the group law on $\mathcal{J}_\mathcal{C}$ states that the cubic polynomial[8]

$$\begin{aligned} l(x) &= E_1(x-u'_P)(x-u_R)(x-u'_R) + E_2(x-u_P)(x-u_R)(x-u'_R) \\ &\quad + E_3(x-u_P)(x-u'_P)(x-u'_R) + E_4(x-u_P)(x-u'_P)(x-u_R) \\ &= (G_1x - G_2)(x^2 + a_1^R x + a_0^R) + (G_3x - G_4)(x^2 + a_1^P x + a_0^P) \end{aligned}$$

satisfies $\ell(x) \equiv b(x) \bmod a(x)$ when $\langle a(x), b(x)\rangle$ is any of P, R or $\ominus(R \oplus P)$. Together with $b(x)^2 \equiv f(x) \pmod{a(x)}$, which is satisfied by every $\langle a(x), b(x)\rangle$ in $\mathcal{J}_\mathcal{C}$, this gives (after some tedious symbolic manipulations, or, alternatively, by Littlewood's principle) the relations (4), (5), and (6). □

The two Kummers are related by a linear projective isomorphism $\tau : \mathcal{K}_\mathcal{C}^{\text{fast}} \xrightarrow{\sim} \mathcal{K}_\mathcal{C}^{\text{gen}}$, which maps $(X : Y : Z : T)$ to $(\xi_1 : \xi_2 : \xi_3 : \xi_4) = (X : Y : Z : T)M_\tau$ where

$$M_\tau = \begin{pmatrix} 1 & \frac{\lambda-\mu\nu}{\lambda-\nu} & \frac{\lambda\nu(1-\mu)}{\lambda-\nu} & \frac{\lambda\nu(\lambda-\mu\nu)}{\lambda-\nu} \\ \frac{a(1-\mu)}{b(\lambda-\nu)} & \frac{a(\lambda-\mu\nu)}{b(\lambda-\nu)} & \frac{a}{b}\mu & \frac{a\mu(\lambda-\mu\nu)}{b(\lambda-\nu)} \\ \frac{a(\mu-\lambda)}{c(\lambda-\nu)} & \frac{a(\mu\nu-\lambda)}{c(\lambda-\nu)} & \frac{a\lambda\mu(\nu-1)}{c(\lambda-\nu)} & \frac{a\lambda\mu(\mu\nu-\lambda)}{c(\lambda-\nu)} \\ \frac{a(\nu-1)}{d(\lambda-\nu)} & \frac{a(\mu\nu-\lambda)}{d(\lambda-\nu)} & \frac{a\nu(\mu-\lambda)}{d(\lambda-\nu)} & \frac{a\nu(\mu\nu-\lambda)}{d(\lambda-\nu)} \end{pmatrix}.$$

The map $\rho \circ \tau : \mathcal{K}_\mathcal{C}^{\text{fast}} \to \mathbb{P}^2$ is defined by the matrix M'_τ formed by the first three columns of M_τ. The inverse isomorphism $\tau^{-1} : \mathcal{K}_\mathcal{C}^{\text{gen}} \to \mathcal{K}_\mathcal{C}^{\text{fast}}$ is defined by any scalar multiple of M_τ^{-1}, and then $\pm P = \tau^{-1}(\pi(P))$ for all P in $\mathcal{J}_\mathcal{C}$.

Proposition 2. *Let P and R be in $\mathcal{J}_\mathcal{C}(\mathbb{F}_q)$. Given $(P, \pm P, \pm R, \pm(R \oplus P))$, Algorithm 2 computes R in* $107\mathtt{M} + 11\mathtt{S} + 4\mathtt{m_c} + 81\mathtt{a} + 1\mathtt{I}$.

Proof. We have $a_1^R = -\xi_2^R$ and $a_0^R = \xi_3^R$; it remains to compute b_1^R and b_0^R using Proposition 1, maintaining the notation of its proof. Let $E := \xi_1^R((DG_1)^2 + (DG_3)^2)$, $\Delta := D^2\left(2\xi_1^\oplus\xi_1^\ominus G_2^2 - (\xi_2^\oplus\xi_1^\ominus + \xi_1^\oplus\xi_2^\ominus)G_1G_2 + (\xi_3^\oplus\xi_1^\ominus + \xi_1^\oplus\xi_3^\ominus)G_1^2\right)$, and $F := -2(\xi_1^R)^2 D\Delta$. Note that $C = F/(\xi_1^R E)$ and $\xi_1^R(DG_3)^2 = \xi_1^R(\xi_4^R D + f_1Z_1Z_2 + f_2Z_2^2 + f_3Z_2Z_3 + f_4Z_3^2) + (\xi_3^R Z_2 + \xi_2^R Z_3)Z_3$. Now, to Algorithm 2: Lines 1-4 compute $\pi(R)$, $\rho(\pi(P \oplus R))$, and $\rho(\pi(P \ominus R))$.[9] Then Lines 5-6 compute $D(G_1, G_2)$ from (b_1^P, b_0^P); Lines 7-8 compute $C(G_3, G_4)$ from $D(G_1, G_2)$; Lines 9-13 compute $F(b_1^P, b_0^P)$ from $EC(G_3, G_4)$. Finally, Lines 14-19 compute F and its inverse and renormalize, yielding R. □

Remark 3. Algorithm 2 assumes that P is not a special point in $\mathcal{J}_\mathcal{C}$, and that $\pm R$ is not the image of a special point $R \in \mathcal{J}_\mathcal{C}$. This assumption is reasonable for all cryptographic intents and purposes, since P is typically an input point to

[8] The cubic curve $y = \ell(x)$ is analogous to the line through P, R, and $\ominus(R \oplus P)$ in the classic elliptic curve group law.

[9] Okeya and Sakurai noticed that the formulæ for y-coordinate recovery on Montgomery curves are simpler if $\pm(R \ominus P)$ is also known [23, pp. 129–130]; here, we take advantage of an analogous simplification in genus 2.

Algorithm 2: `Recover`: Recovery from $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$ to $\mathcal{J}_{\mathcal{C}}$.

Input: $(P, \pm P, \pm R, \pm(R \oplus P)) \in \mathcal{J}_{\mathcal{C}} \times (\mathcal{K}_{\mathcal{C}}^{\text{fast}})^3$ for P and (unknown) R in $\mathcal{J}_{\mathcal{C}}$.
Output: $R \in \mathcal{J}_{\mathcal{C}}$.
Cost: $107\mathtt{M} + 11\mathtt{S} + 4\mathtt{m_c} + 81\mathtt{a} + 1\mathtt{I}$, assuming precomputed M_τ.

1 $\pm(R \ominus P) \leftarrow \mathtt{xADD}^*(\pm R, \pm P, \pm(R \oplus P))$ // 14M+8S+4m_c+24a
2 $(\xi_1^R : \xi_2^R : \xi_3^R : \xi_4^R) \leftarrow \pm R \cdot M_\tau$ // 15M+12a
3 $(\xi_1^\oplus : \xi_2^\oplus : \xi_3^\oplus) \leftarrow \pm(R \oplus P) \cdot M'_\tau$ // 11M+9a
4 $(\xi_1^\ominus : \xi_2^\ominus : \xi_3^\ominus) \leftarrow \pm(R \ominus P) \cdot M'_\tau$ // 11M+9a
5 $(\mathsf{Z}_1, \mathsf{Z}_2, \mathsf{Z}_3) \leftarrow (a_1^P \cdot \xi_1^R + \xi_2^R, \xi_3^R - a_0^P \cdot \xi_1^R, -(a_0^P \cdot \xi_2^R + a_1^P \cdot \xi_3^R))$ // 4M+4a
6 $(\mathsf{DG}_1, \mathsf{DG}_2) \leftarrow (\mathsf{Z}_2 \cdot b_1^P + \mathsf{Z}_1 \cdot b_0^P, (\mathsf{Z}_1 \cdot a_0^P \cdot b_1^P - \mathsf{Z}_1 \cdot a_1^P + \mathsf{Z}_2) \cdot b_0^P)$ // 6M+3a
7 $(\mathsf{Y}_{13}, \mathsf{Y}_{21}, \mathsf{Y}_{23}) \leftarrow (\xi_1^\oplus \cdot \xi_3^\ominus - \xi_3^\oplus \cdot \xi_1^\ominus, \xi_2^\oplus \cdot \xi_1^\ominus - \xi_1^\oplus \cdot \xi_2^\ominus, \xi_2^\oplus \cdot \xi_3^\ominus - \xi_3^\oplus \cdot \xi_2^\ominus)$ // 6M+3a
8 $(\mathsf{CG}_3, \mathsf{CG}_4) \leftarrow (\mathsf{DG}_1 \cdot \mathsf{Y}_{13} + \mathsf{DG}_2 \cdot \mathsf{Y}_{21}, \mathsf{DG}_1 \cdot \mathsf{Y}_{23} - \mathsf{DG}_2 \cdot \mathsf{Y}_{13})$ // 4M+2a
9 $\mathsf{xiD} \leftarrow \mathsf{Z}_2^2 - \mathsf{Z}_1 \cdot \mathsf{Z}_3$ // 1M+1S+1a
10 $\mathsf{E} \leftarrow \xi_1^R \cdot ((f_3 \cdot \mathsf{Z}_3 + f_2 \cdot \mathsf{Z}_2 + f_1 \cdot \mathsf{Z}_1) \cdot \mathsf{Z}_2 + \mathsf{DG}_1^2) + \xi_4^R \cdot \mathsf{xiD}$ // 6M+1S+4a
11 $\mathsf{E} \leftarrow \mathsf{E} + \mathsf{Z}_3 \cdot (\mathsf{Z}_3 \cdot (f_4 \cdot \xi_1^R + \xi_2^R) + \mathsf{Z}_2 \cdot \xi_3^R)$ // 4M+3a
12 $\mathsf{xiFb}_1 \leftarrow \mathsf{E} \cdot ((\mathsf{Z}_1 \cdot \xi_2^R - \mathsf{Z}_2 \cdot \xi_1^R) \cdot \mathsf{CG}_3 - \mathsf{Z}_1 \cdot \xi_3^R \cdot \mathsf{CG}_4)$ // 6M+2a
13 $\mathsf{xiFb}_0 \leftarrow \mathsf{E} \cdot (\mathsf{Z}_2 \cdot \xi_1^R \cdot \mathsf{CG}_4 - \mathsf{Z}_1 \cdot \xi_1^R \cdot \mathsf{CG}_3)$ // 5M+1a
14 $\mathsf{Delta} \leftarrow \mathsf{DG}_1 \cdot \big(\mathsf{CG}_3 + 2\xi_1^\oplus \cdot (\mathsf{DG}_1 \cdot \xi_3^\ominus + \mathsf{DG}_2 \cdot \xi_2^\ominus)\big) + 2\mathsf{DG}_2^2 \cdot \xi_1^\oplus \cdot \xi_1^\ominus$ // 6M+1S+5a
15 $\mathsf{F} \leftarrow -2\mathsf{xiD} \cdot \xi_1^R \cdot \mathsf{Delta}$ // 2M+2a
16 $\mathsf{invxiF} \leftarrow 1/(\mathsf{F} \cdot \xi_1^R)$ // 1M + 1I
17 $\mathsf{invxi} \leftarrow \mathsf{F} \cdot \mathsf{invxiF}$ // 1M
18 $(a_1^R, a_0^R, b_1^R, b_0^R) \leftarrow (-\mathsf{invxi} \cdot \xi_2^R, \mathsf{invxi} \cdot \xi_3^R, \mathsf{invxiF} \cdot \mathsf{xiFb}_1, \mathsf{invxiF} \cdot \mathsf{xiFb}_0)$ // 4M+1a
19 **return** $\langle x^2 + a_1^R x + a_0^R, b_1^R x + b_0^R \rangle$

a scalar multiplication routine (that, if special, can be detected and rejected), and R is a secret multiple of P (that will be special with negligible probability). For completeness, we note that if either or both of P or R is special, then we can still use Algorithm 2 by translating the input points by a well-chosen 2-torsion point, and updating the output appropriately by the same translation (we recall that on the fast Kummer, all 16 of the two-torsion points are rational, which gives us plenty of choice here). A fully-fledged implementation could be made to run in constant-time (for *all* input and output points) by always performing these translations and choosing the correct inputs and outputs using bitmasks.

Remark 4. Gaudry computes the preimages in $\mathcal{J}_\mathcal{C}$ for points in $\mathcal{K}_\mathcal{C}^{\text{fast}}$ in [15, §4.3]; but this method (which is analogous to computing (x, y) and $(x, -y)$ on an elliptic curve given x and $y^2 = x^3 + ax + b$) cannot tell us which of the two preimages is the correct image for a given scalar multiplication on $\mathcal{J}_\mathcal{C}$.

4 Uniform one-dimensional scalar multiplication

We are finally ready for scalar multiplication. Algorithm 3 lifts the Montgomery ladder [22] pseudomultiplication $(m, \pm P) \mapsto \pm[m]P$ on $\mathcal{K}_\mathcal{C}^{\text{fast}}$ to a full scalar multiplication $(m, P) \mapsto [m]P$ on $\mathcal{J}_\mathcal{C}$, generalizing the methods of [20], [23], and [4]. It is visibly uniform with respect to (fixed-length) m.

Algorithm 3: One-dimensional uniform scalar multiplication on $\mathcal{J}_\mathcal{C}$ via `Project`, the Montgomery ladder, and `Recover`

Input: An integer $m = \sum_{i=0}^{\beta-1} m_i 2^i \geq 0$, with $m_{\beta-1} \neq 0$; a point $P \in \mathcal{J}_\mathcal{C}(\mathbb{F}_q) \setminus \mathcal{B}$
Output: $[m]P$
Cost: $(7\beta + 115)\mathtt{M} + (12\beta + 8)\mathtt{S} + (12\beta + 4)\mathtt{m_c} + (32\beta + 79)\mathtt{a} + 2\mathtt{I}$

1 $\pm P \leftarrow \mathtt{Project}(P)$ // 8M+1S+4m_c+14a
2 $\mathtt{xP} \leftarrow \mathtt{Wrap}(\pm P)$ // 7M+1I
3 $(\mathtt{t}_1, \mathtt{t}_2) \leftarrow (\pm P, \mathtt{xDBL}(\pm P))$ // 8S+8m_c+16a
4 **for** $i = \beta - 2$ *down to* 0 **do**
5 $\quad (\mathtt{t}_1, \mathtt{t}_2) \leftarrow \mathtt{SWAP}(m_i, (\mathtt{t}_1, \mathtt{t}_2))$
6 $\quad (\mathtt{t}_1, \mathtt{t}_2) \leftarrow \mathtt{xDBLADD}(\mathtt{t}_1, \mathtt{t}_2, \mathtt{xP})$ // 7M+12S+12m_c+32a
7 $\quad (\mathtt{t}_1, \mathtt{t}_2) \leftarrow \mathtt{SWAP}(m_i, (\mathtt{t}_1, \mathtt{t}_2))$
8 **end**
9 **return** $\mathtt{Recover}(P, \pm P, \mathtt{t}_1, \mathtt{t}_2)$ // 107M+11S+4m_c+81a+1I

Theorem 1 (`Project` + Montgomery ladder + `Recover`). *Let $m > 0$ be a β-bit integer, and P a point in $\mathcal{J}_\mathcal{C}(\mathbb{F}_q)$. Algorithm 3 computes $[m]P$ using one* `Project`*, one* `Wrap`*, one* `xDBL`*, $\beta - 1$* `xDBLADD`*s, and one* `Recover`*; that is, in $(7\beta + 115)\mathtt{M} + (12\beta + 8)\mathtt{S} + (12\beta + 4)\mathtt{m_c} + (32\beta + 79)\mathtt{a} + 2\mathtt{I}$.*

Proof. Lines 3-7 are the Montgomery ladder; after each of the $\beta - 1$ iterations we have $t_1 = \pm[\lfloor m/2^i \rfloor]P$ and $t_2 = \pm[\lfloor m/2^i \rfloor + 1]P$, so $(t_1, t_2) = (\pm[m]P, \pm[m+1]P)$ at Line 8, and $\mathtt{Recover}(P, \pm P, t_1, t_2) = [m]P$. □

If the base point P is fixed then we can precompute Lines 1-3 in Algorithm 3, thus saving $15\mathtt{M} + 9\mathtt{S} + 10\mathtt{m_c} + 30\mathtt{a} + 1\mathtt{I}$ in subsequent calls.

5 Uniform two-dimensional scalar multiplication

Algorithm 4 defines a uniform two-dimensional scalar multiplication for computing $[m]P \oplus [n]Q$, where P and Q (and $P \oplus Q$ and $P \ominus Q$) are in $\mathcal{J}_{\mathcal{C}} \setminus \mathcal{B}$ and $m = \sum_{i=0}^{\beta-1} m_i 2^i$ and $n = \sum_{i=0}^{\beta-1} n_i 2^i$ are β-bit scalars (with $m_{\beta-1}$ and/or $n_{\beta-1}$ not zero). The inner pseudomultiplication on $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$ is based on Bernstein's binary differential addition chain [2, §4].[10] It is visibly uniform with respect to (fixed-length) multiscalars (m, n); while this is unnecessary for signature verification, where multiscalars are public, it is useful for GLV-style endomorphism-accelerated scalar multiplication with secret scalars.

Recall the definition of Bernstein's chain: for each pair of non-negative integers (A, B), we have two differential chains $C_0(A, B)$ and $C_1(A, B)$ with

$$C_0(0,0) = C_1(0,0) := ((0,0),(1,0),(0,1),(1,-1)) \ ,$$

and then defined mutually recursively for $A \neq 0$ and/or $B \neq 0$ by

$$C_D(A,B) := C_d(\lfloor A/2 \rfloor, \lfloor B/2 \rfloor) \mid\mid (O, E, M)$$

where $||$ is concatenation, $d = (D+1)(A - \lfloor A/2 \rfloor + 1) + D(B - \lfloor B/2 \rfloor) \pmod 2$, and O, E, and M (the "odd", "even", and "mixed" pairs) are

$$O := (A + (A+1 \bmod 2), B + (B+1 \bmod 2)) \ , \tag{7}$$
$$E := (A + (A+0 \bmod 2), B + (B+0 \bmod 2)) \ , \tag{8}$$
$$M := (A + (A+D \bmod 2), B + (B+D+1 \bmod 2)) \ . \tag{9}$$

By definition, (O, E, M) contains three of the four pairs (A, B), $(A+1, B)$, $(A, B+1)$, and $(A+1, B+1)$; the missing pair is $(A + (A+D+1 \bmod 2), B + (B+D \bmod 2))$. The differences $M - O$, $M - E$, and $O - E$ depend only on D and the parities of A and B, as shown in Table 2.

Table 2. The differences between M, O, and E as functions of D and $A, B \pmod 2$.

$A \pmod 2$	$B \pmod 2$	$O - E$	$M - O$	$M - E$
0	0	$(1, 1)$	$(D-1, -D)$	$(D, 1-D)$
0	1	$(1, -1)$	$(D-1, D)$	$(D, D-1)$
1	0	$(-1, 1)$	$(1-D, -D)$	$(-D, 1-D)$
1	1	$(-1, -1)$	$(1-D, D)$	$(-D, D-1)$

[10] The elliptic curve x-line version of this pseudomultiplication was used in [12].

Algorithm 4: Two-dimensional uniform scalar multiplication on $\mathcal{J}_{\mathcal{C}}$ via `Project`, Bernstein's two-dimensional "binary" differential addition chain, and `Recover`.

Input: $m = \sum_{i=0}^{\beta-1} m_i 2^i$ and $n = \sum_{i=0}^{\beta-1} n_i 2^i$ with $m_{\beta-1} n_{\beta-1} \neq 0$;
$P, Q \in \mathcal{J}_{\mathcal{C}}(\mathbb{F}_q) \setminus \mathcal{B}$ such that $P \oplus Q \notin \mathcal{B}$ and $P \ominus Q \notin \mathcal{B}$

Output: $[m]P \oplus [n]Q$

Cost: $(14\beta + 203)\mathtt{M} + (20\beta + 16)\mathtt{S} + (16\beta + 16)\mathtt{m_c} + (56\beta + 138)\mathtt{a} + 3\mathtt{I}$

$S \leftarrow \mathtt{JacADD}(P, Q)$ // 28M+2S+35a+1I

$(\pm P, \pm Q, \pm S) \leftarrow (\mathtt{Project}(P), \mathtt{Project}(Q), \mathtt{Project}(S))$ // 24M+3S+12$\mathtt{m_c}$+42a

$\pm D \leftarrow \mathtt{xADD}^*(\pm P, \pm Q, \pm S)$ // 14M+8S+4$\mathtt{m_c}$+24a

$(\mathsf{xP}, \mathsf{xQ}, \mathsf{xS}, \mathsf{xD}) \leftarrow \mathtt{Wrap4}(\pm P, \pm Q, \pm S, \pm D)$ // 37M+1I

$\mathsf{d}_0 \leftarrow m_0$

for $i \leftarrow 1$ *up to* $\beta - 1$ **do** $\mathsf{d}_i \leftarrow \mathsf{d}_{i-1} + (\mathsf{d}_{i-1} + 1)(m_{i-1} + m_i) + \mathsf{d}_{i-1}(n_{i-1} + n_i)$

$\mathsf{U}_0 \leftarrow \mathtt{SELECT}(n_{\beta-1}, (\mathsf{xP}, \mathsf{xQ}))$

$\mathsf{U}_1 \leftarrow \mathtt{SELECT}(m_{\beta-1} n_{\beta-1}, (\mathsf{U}_0, \mathsf{xS}))$

$(\mathsf{U}_2, \mathsf{U}_3) \leftarrow \mathtt{SWAP}(\mathsf{d}_{\beta-1}, (\mathsf{xP}, \mathsf{xQ}))$

$(\mathsf{U}_4, \mathsf{U}_5) \leftarrow \mathtt{SELECT}(\mathsf{d}_{\beta-1}(m_{\beta-1} + n_{\beta-1}) + m_{\beta-1} + 1, ((\mathsf{xP}, \mathsf{U}_3), (\mathsf{xQ}, \mathsf{xD})))$

$(\mathsf{U}_6, \mathsf{U}_7) \leftarrow \mathtt{SELECT}(m_{\beta-1}(n_{\beta-1} + 1), ((\mathsf{xS}, \mathsf{U}_2), (\mathsf{U}_4, \mathsf{xS})))$

$(\mathsf{E}_{\beta-1}, \mathsf{U}_8) \leftarrow \mathtt{xDBLADD}(\mathsf{U}_1, \mathsf{U}_7, \mathsf{U}_5)$ // 7M+12S+12$\mathtt{m_c}$+32a

$(\mathsf{O}_{\beta-1}, \mathsf{M}_{\beta-1}) \leftarrow \mathtt{SWAP}(\mathsf{d}_{\beta-1}(m_{\beta-1} + n_{\beta-1}) + m_{\beta-1} + 1, (\mathsf{U}_6, \mathsf{U}_8))$

for $i \leftarrow \beta - 2$ *down to* 0 **do**

  $\mathsf{O}_i \leftarrow \mathtt{xADD}(\mathsf{O}_{i+1}, \mathsf{E}_{i+1}, \mathtt{SELECT}(m_i + n_i, (\mathsf{xS}, \mathsf{xD})))$ // 7M+8S+4$\mathtt{m_c}$+24a

  $\mathsf{V}_0 \leftarrow \mathtt{SELECT}((d_i + 1)(m_{i+1} + m_i) + d_i(n_{i+1} + n_i), (\mathsf{O}_{i+1}, \mathsf{E}_{i+1}))$

  $(\mathsf{V}_1, \mathsf{V}_2) \leftarrow \mathtt{SWAP}(m_i + m_{i+1} + n_i + n_{i+1}, (\mathsf{V}_0, \mathsf{M}_{i+1}))$

  $(\mathsf{E}_i, \mathsf{M}_i) \leftarrow \mathtt{xDBLADD}(\mathsf{V}_1, \mathsf{V}_2, \mathtt{SELECT}(\mathsf{d}_i, (\mathsf{xP}, \mathsf{xQ})))$ // 7M+12S+12$\mathtt{m_c}$+32a

end

$(\mathsf{W}_0, \mathsf{W}_1) \leftarrow \mathtt{SWAP}(m_0, (\mathsf{O}_0, \mathsf{E}_0))$

$(\mathsf{W}_2, \mathsf{W}_3, \mathsf{W}_4, \mathsf{W}_5) \leftarrow \mathtt{SELECT}(m_0 + n_0, ((S, \mathsf{xS}, \mathsf{W}_0, \mathsf{W}_1), (P, \mathsf{xP}, \mathsf{M}_0, \mathsf{W}_0)))$

return $\mathtt{Recover}(\mathsf{W}_2, \mathsf{W}_3, \mathsf{W}_4, \mathsf{W}_5)$ // 107M+11S+4$\mathtt{m_c}$+81a+1I

Table 3. The state of Algorithm 4 after the main loop.

(m_0, n_0)	O_0	E_0	M_0 if $d_0 = 0$	M_0 if $d_0 = 1$	$R = [m]P \oplus [n]Q$
$(0,0)$	$\pm(R \oplus S)$	$\pm R$	$\pm(R \oplus Q)$	$\pm(R \oplus P)$	$\texttt{Recover}(S, \pm S, E_0, O_0)$
$(0,1)$	$\pm(R \oplus P)$	$\pm(R \oplus Q)$	$\pm R$	$\pm(R \oplus S)$	$\texttt{Recover}(P, \pm P, M_0, O_0)$
$(1,0)$	$\pm(R \oplus Q)$	$\pm(R \oplus P)$	$\pm(R \oplus S)$	$\pm R$	$\texttt{Recover}(P, \pm P, M_0, E_0)$
$(1,1)$	$\pm R$	$\pm(R \oplus S)$	$\pm(R \oplus P)$	$\pm(R \oplus S)$	$\texttt{Recover}(S, \pm S, O_0, E_0)$

Theorem 2 (`Project` + Bernstein's binary chain + `Recover`). *Let P and Q be in $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q)$; let m and n be positive integers, with β the bitlength of* $\max(m, n)$. *Algorithm 4 computes $[m]P \oplus [n]Q$ using one* `JacADD`, *three* `Project`*s, one* `Wrap4`, *one* `xADD`*, $\beta - 1$ `xADD`*s, β* `xDBLADD`*s, and one* `Recover`*; that is,* $(14\beta + 203)\mathtt{M} + (20\beta + 16)\mathtt{S} + (16\beta + 16)\mathtt{m_c} + (56\beta + 138)\mathtt{a} + 3\mathtt{I}$.

Proof. Consider $C_{m_0}(m, n) = C_0(0,0)||(O_{\beta-1}, E_{\beta-1}, M_{\beta-1})|| \cdots ||(O_0, E_0, M_0)$. It follows from (7), (8), and (9) that (m, n) is one of O_0, E_0, or M_0 (and parity tells us which one). On the other hand, we have

$$C_{d_i}(\lfloor m/2^i \rfloor, \lfloor n/2^i \rfloor) = C_{d_{i+1}}(\lfloor m/2^{i+1} \rfloor, \lfloor n/2^{i+1} \rfloor) \;||\; (O_i, E_i, M_i) \tag{10}$$

for $0 \le i \le \beta - 2$, where the bits d_i are defined by $d_0 = m_0$ and $d_i := d_{i-1} + (d_{i-1} + 1)(m_{i-1} + m_i) + d_{i-1}(n_{i-1} + n_i)$ for $i > 0$. The definition of the chains, Table 2, and considerations of parity yield the following relations which allow us to construct each triple (O_i, E_i, M_i) from its antecedent $(O_{i+1}, E_{i+1}, M_{i+1})$:

1. $O_i = O_{i+1} + E_{i+1}$, with $O_{i+1} - E_{i+1} = \pm(1,1)$ if $m_i = n_i$ and $\pm(1,-1)$ if $m_i \neq n_i$.
2. $E_i = 2E_{i+1}$ if $(m_i, n_i) = (m_{i+1}, n_{i+1})$; or $2O_{i+1}$ if $m_{i+1} \neq m_i$ and $n_{i+1} \neq n_i$; or $2M_{i+1}$ otherwise.
3. If $d_i = 0$ then $M_i = M_{i+1} + X$, where $X = E_{i+1}$ if $m_{i+1} = m_i$, or O_{i+1} if $m_{i+1} \neq m_i$; and $M_{i+1} - X = \pm(0,1)$.
4. If $d_i = 1$ then $M_i = M_{i+1} + X$, where $X = E_{i+1}$ if $n_{i+1} \neq n_i$, or O_{i+1} if $n_{i+1} = n_i$; and $M_{i+1} - X = \pm(1,0)$.

We can therefore compute $\pm R = \pm([m]P \oplus [n]Q)$ by mapping each pair (a, b) in $C_{m_0}(m, n)$ to $\pm([a]P \oplus [b]Q)$. Lines 1–4 (pre)compute the required difference points $\pm P$, $\pm Q$, $\pm S = \pm(P \oplus Q)$, and $\pm D = \pm(P \ominus Q)$. Lines 5–6 compute all of the d_i. After initializing the first nontrivial segment $(O_{\beta-1}, E_{\beta-1}, M_{\beta-1})$ in Lines 7–13, the main loop (Lines 14–18) derives the following segments using the rules above. Table 3 gives the state of the final segment (O_0, E_0, M_0) immediately after the loop. In each case, we can recover $[m]P \oplus [n]Q$ using the call to `Recover` specified by the corresponding row, as is done in Lines 19–21. □

If the points P and Q are fixed then we can precompute Lines 1-4 in Algorithm 4, thus saving $103\mathtt{M} + 13\mathtt{S} + 16\mathtt{m_c} + 101\mathtt{a} + 2\mathtt{I}$ in subsequent calls.

Remark 5. There are faster two-dimensional differential addition chains that are non-uniform, such as Montgomery's PRAC algorithm [27, Ch. 3], which might

be preferred in scenarios where the multiscalars are not secret (such as signature verification). However, PRAC is not well-suited to our recovery technique, because its outputs do not "differ" by an element with known preimage in $\mathcal{J}_{\mathcal{C}}$.

References

1. Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. 2
2. Daniel J. Bernstein. Differential addition chains. preprint, 2006. 2, 11
3. Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Peter Schwabe. Kummer strikes back: New DH speed records. In Sarkar and Iwata [25], pages 317–337. 3
4. Eric Brier and Marc Joye. Weierstraß elliptic curves and side-channel attacks. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography, 5th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002, Paris, France, February 12-14, 2002, Proceedings*, volume 2274 of *Lecture Notes in Computer Science*, pages 335–345. Springer, 2002. 2, 10
5. Daniel R. L. Brown. Multi-dimensional montgomery ladders for elliptic curves. 2006.http://eprint.iacr.org/2006/220. 3
6. David G. Cantor. Computing in the Jacobian of a hyperelliptic curve. *Mathematics of computation*, 48(177):95–101, 1987. 3
7. J. W. S. Cassels and E. V. Flynn. *Prolegomena to a middlebrow arithmetic of curves of genus 2*, volume 230. Cambridge University Press, 1996. 6, 7
8. David V. Chudnovsky and Gregory V. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Adv. in Appl. Math.*, 7:385–434, 1986. 5
9. Ping Ngai Chung, Craig Costello, and Benjamin Smith. Fast, uniform, and compact scalar multiplication for elliptic curves and genus 2 jacobians with applications to signature schemes. IACR Cryptology ePrint Archive, report 2015/983, 2015. 1, 3, 4
10. Romain Cosset. *Applications of theta functions for hyperelliptic curve cryptography.* Ph.D Thesis, Université Henri Poincaré - Nancy I, November 2011. 5
11. Romain Cosset and Christophe Arene. Construction of a k-complete addition law on jacobians of hyperelliptic curves of genus two. *Contemporary Mathematics*, 574:1–14, 2012. 3
12. Craig Costello, Hüseyin Hisil, and Benjamin Smith. Faster compact Diffie–Hellman: Endomorphisms on the x-line. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2014. 11
13. Sylvain Duquesne. Montgomery scalar multiplication for genus 2 curves. In Duncan A. Buell, editor, *Algorithmic Number Theory, 6th International Symposium, ANTS-VI, Burlington, VT, USA, June 13-18, 2004, Proceedings*, volume 3076 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2004. 6

14. Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200. Springer, 2001. 3
15. Pierrick Gaudry. Fast genus 2 arithmetic based on Theta functions. *J. Mathematical Cryptology*, 1(3):243–265, 2007. 1, 5, 10, 16
16. Pierrick Gaudry and Eric Schost. Genus 2 point counting over prime fields. *J. Symb. Comput.*, 47(4):368–400, 2012. 5
17. Huseyin Hisil and Craig Costello. Jacobian coordinates on genus 2 curves. In Sarkar and Iwata [25], pages 338–357. 3, 4
18. Tetsuya Izu and Tsuyoshi Takagi. Exceptional procedure attack on elliptic curve cryptosystems. In Yvo Desmedt, editor, *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, volume 2567 of *Lecture Notes in Computer Science*, pages 224–239. Springer, 2003. 3
19. David Kohel. Arithmetic of split kummer surfaces: Montgomery endomorphism of edwards products. In Yeow Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing, editors, *Coding and Cryptology - Third International Workshop, IWCC 2011, Qingdao, China, May 30-June 3, 2011. Proceedings*, volume 6639 of *Lecture Notes in Computer Science*, pages 238–245. Springer, 2011. 3
20. Julio López and Ricardo Dahab. Fast multiplication on elliptic curves over GF(2^m) without precomputation. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 316–327. Springer, 1999. 2, 10
21. David Lubicz and Damien Robert. Arithmetic on abelian and kummer varieties. *Finite Fields and Their Applications*, 39:130–158, 2016. 3
22. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987. 2, 10
23. Katsuyuki Okeya and Kouichi Sakurai. Efficient elliptic curve cryptosystems from a scalar multiplication algorithm with recovery of the y-coordinate on a Montgomery-form elliptic curve. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2001. 2, 8, 10
24. Joost Renes, Peter Schwabe, Benjamin Smith, and Lejla Batina. μKummer: Efficient hyperelliptic signatures and key exchange on microcontrollers. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 301–320, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. 3, 5
25. Palash Sarkar and Tetsu Iwata, editors. *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*. Springer, 2014. 14, 15
26. Nigel P. Smart and Samir Siksek. A fast Diffie–Hellman protocol in genus 2. *Journal of cryptology*, 12(1):67–73, 1999. 2, 6

27. Martijn Stam. *Speeding up subgroup cryptosystems.* Technische Universiteit Eindhoven, 2003. 13

A Fast Kummer arithmetic

We recall the formulæ for operations on fast Kummers from [15, §3.2]. To simplify the presentation of our algorithms, we define three operations on points in $\mathbb{P}^3$ (or more precisely, on 4-tuples of elements of $\mathbb{F}_q$). First, $\mathcal{M} : \mathbb{P}^3 \times \mathbb{P}^3 \to \mathbb{P}^3$ multiplies the corresponding coordinates of a pair of points:

$$\mathcal{M} : ((x_1 : y_1 : z_1 : t_1), (x_2 : y_2 : z_2 : t_2)) \longmapsto (x_1x_2 : y_1y_2 : z_1z_2 : t_1t_2) \ ,$$

costing 4M. The special case $(x_1 : y_1 : z_1 : t_1) = (x_2 : y_2 : z_2 : t_2)$ is denoted by

$$\mathcal{S} : (x : y : z : t) \longmapsto (x^2 : y^2 : z^2 : t^2) \ ,$$

costing 4S. Finally, the Hadamard transform[11] is defined by

$$\mathcal{H} : (x : y : z : t) \longmapsto (x' : y' : z' : t') \quad \text{where} \quad \begin{cases} x' = x + y + z + t \ , \\ y' = x + y - z - t \ , \\ z' = x - y + z - t \ , \\ t' = x - y - z + t \ . \end{cases}$$

The Hadamard transform can easily be implemented with 8a.

The basic (unoptimized) pseudo-addition operation is xADD* (Algorithm 5). The pseudo-doubling operation is xDBL (Algorithm 6).

Algorithm 5: xADD*: Differential addition on $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$.

Input: $(\pm P, \pm Q, (x_\ominus : y_\ominus : z_\ominus : t_\ominus) = \pm(P \ominus Q))$ for some P, Q in $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q)$ with $P \ominus Q \notin \mathcal{B}$.
Output: $\pm(P \oplus Q) \in \mathcal{K}_{\mathcal{C}}^{\text{fast}}$.
Cost: $14\mathtt{M} + 8\mathtt{S} + 4\mathtt{m_c} + 24\mathtt{a}$

1 $(\mathsf{V}_1, \mathsf{V}_2) \leftarrow (\mathcal{H}(\mathcal{S}(\pm P)), \mathcal{H}(\mathcal{S}(\pm Q)))$ // 8S+16a
2 $\mathsf{V}_3 \leftarrow \mathcal{M}(\mathsf{V}_1, \mathsf{V}_2)$ // 4M
3 $\mathsf{V}_4 \leftarrow \mathcal{H}(\mathcal{M}(\mathsf{V}_3, (1/A : 1/B, 1/C, 1/D))$ // 4m_c+8a
4 $(\mathsf{C}_1, \mathsf{C}_2) \leftarrow (x_\ominus \cdot y_\ominus, z_\ominus \cdot t_\ominus)$ // 2M
5 **return** $\mathcal{M}(\mathsf{V}_4, (y_\ominus \cdot \mathsf{C}_2, x_\ominus \cdot \mathsf{C}_2, t_\ominus \cdot \mathsf{C}_1, z_\ominus \cdot \mathsf{C}_1))$ // 8M

Lines 4 and 5 of Algorithm 5 compute the point $(y_\ominus z_\ominus t_\ominus : x_\ominus z_\ominus t_\ominus : x_\ominus y_\ominus t_\ominus : x_\ominus y_\ominus z_\ominus)$, which is projectively equivalent to $(1/x_\ominus : 1/y_\ominus : 1/z_\ominus : 1/t_\ominus)$, but requires no inversions (note that this is generally *not* a point on $\mathcal{K}_{\mathcal{C}}$). This is the only point in our pseudoarithmetic where the third argument $(x_\ominus : y_\ominus : z_\ominus : t_\ominus)$ appears. In practice, the pseudoadditions used in our scalar multiplication all

[11] Note $(A : B : C : D) = \mathcal{H}((a : b : c : d))$; dually, $(a : b : c : d) = \mathcal{H}((A : B : C : D))$.

Algorithm 6: `xDBL`: Pseudo-doubling on $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$.

Input: $\pm P$ in $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$ for P in $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q)$.
Output: $\pm[2]P$
Cost: $8\mathtt{S} + 8\mathtt{m_c} + 16\mathtt{a}$

1 $\mathsf{V}_1 \leftarrow \mathcal{H}(\mathcal{S}(\pm P))$ // 4S+8a
2 $\mathsf{V}_2 \leftarrow \mathcal{S}(\mathsf{V}_1)$ // 4S
3 $\mathsf{V}_3 \leftarrow \mathcal{H}(\mathcal{M}(\mathsf{V}_2, (1/A : 1/B : 1/C : 1/D)))$ // $4\mathtt{m_c}$+8a
4 **return** $\mathcal{M}(\mathsf{V}_4, (1/a : 1/b : 1/c : 1/d))$ // $4\mathtt{m_c}$

use a fixed third argument, so it makes sense to precompute this "inverted" point and to scale it by $x_\ominus$ so that the first coordinate is 1, thus saving $7\mathbf{M}$ in each subsequent pseudo-addition for a one-off cost of $1\mathbf{I}$. The resulting data can be stored as the 3-tuple $(x_\ominus/y_\ominus, x_\ominus/z_\ominus, x_\ominus/t_\ominus)$, ignoring the trivial first coordinate: this is the *wrapped* form of $\pm(P \ominus Q)$. The function `Wrap` (Algorithm 7) applies this transformation; we also include `Wrap4` (Algorithm 8), which simultaneously `Wrap`s four points using a single shared inversion.

Algorithm 7: `Wrap`: $(x : y : z : t) \mapsto (x/y, x/z, x/t)$.

Input: $(x_P : y_P : z_P : t_P) = \pm P$ for P in $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q) \setminus \mathcal{B}$
Output: $(x/y, x/z, x/t) \in \mathbb{F}_q^3$.
Cost: $7\mathtt{M} + 1\mathtt{I}$

1 $\mathsf{V}_1 \leftarrow y \cdot z$ // 1M
2 $\mathsf{V}_2 \leftarrow x/(\mathsf{V}_1 \cdot t)$ // 2M+1I
3 $\mathsf{V}_3 \leftarrow \mathsf{V}_2 \cdot t$ // 1M
4 **return** $(\mathsf{V}_3 \cdot z, \mathsf{V}_3 \cdot y, \mathsf{V}_1 \cdot \mathsf{V}_2)$ // 3M

We can now define `xADD` (Algorithm 9), an optimized pseudo-addition using a `Wrap`ped third argument, and `xDBLADD` (Algorithm 10), which is an optimized combined pseudo-doubling-and-addition.

Algorithm 8: `Wrap4`: four simultaneous Kummer point wrappings

Input: $(\pm P, \pm Q, \pm S, \pm D)$ for P, Q, S, D in $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q) \setminus \mathcal{B}$
Output: $\mathtt{Wrap}(\pm P), \mathtt{Wrap}(\pm Q), \mathtt{Wrap}(\pm S), \mathtt{Wrap}(\pm D)$
Cost: $37\mathtt{M} + 1\mathtt{I}$

1 $(\mathsf{c}_1, \mathsf{c}_2, \mathsf{c}_3, \mathsf{c}_4) \leftarrow (y^P \cdot z^P, y^Q \cdot z^Q, y^S \cdot z^S, y^D \cdot z^D)$ // 4M
2 $(\mathsf{f}_1, \mathsf{f}_2, \mathsf{f}_3, \mathsf{f}_4) \leftarrow (\mathsf{c}_1 \cdot t^P, \mathsf{c}_2 \cdot t^Q, \mathsf{c}_3 \cdot t^S, \mathsf{c}_4 \cdot t^D)$ // 4M
3 $(\mathsf{g}_1, \mathsf{g}_2) \leftarrow (\mathsf{f}_1 \cdot \mathsf{f}_2, \mathsf{f}_3 \cdot \mathsf{f}_4)$ // 2M
4 $\mathsf{I} \leftarrow 1/(\mathsf{g}_1 \cdot \mathsf{g}_2)$ // 1M+1I
5 $(\mathsf{h}_1, \mathsf{h}_2) \leftarrow (\mathsf{g}_1 \cdot \mathsf{I}, \mathsf{g}_2 \cdot \mathsf{I})$ // 2M
6 $(\mathsf{e}_1, \mathsf{e}_2, \mathsf{e}_3, \mathsf{e}_4) \leftarrow (x^P \cdot \mathsf{f}_2 \cdot \mathsf{h}_2, x^Q \cdot \mathsf{f}_1 \cdot \mathsf{h}_2, x^S \cdot \mathsf{f}_4 \cdot \mathsf{h}_1, x^D \cdot \mathsf{f}_3 \cdot \mathsf{h}_1)$ // 8M
7 $(\mathsf{r}_1, \mathsf{r}_2, \mathsf{r}_3, \mathsf{r}_4) \leftarrow (\mathsf{e}_1 \cdot t^P, \mathsf{e}_2 \cdot t^Q, \mathsf{e}_3 \cdot t^S, \mathsf{e}_4 \cdot t^D)$ // 4M
8 **return** $(\mathsf{r}_1 \cdot z^P, \mathsf{r}_1 \cdot y^P, \mathsf{c}_1 \cdot \mathsf{e}_1)$, $(\mathsf{r}_2 \cdot z^Q, \mathsf{r}_2 \cdot y^Q, \mathsf{c}_2 \cdot \mathsf{e}_2)$, $(\mathsf{r}_3 \cdot z^S, \mathsf{r}_3 \cdot y^S, \mathsf{c}_3 \cdot \mathsf{e}_3)$, $(\mathsf{r}_4 \cdot z^D, \mathsf{r}_4 \cdot y^D, \mathsf{c}_4 \cdot \mathsf{e}_4)$ // 12M

Algorithm 9: `xADD`: Differential addition on $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$ with wrapped difference.

Input: $(\pm P, \pm Q, (x_\ominus/y_\ominus, x_\ominus/z_\ominus, x_\ominus/t_\ominus) = \mathtt{Wrap}(\pm(P \ominus Q)))$ for P, Q in $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q)$ with $P \ominus Q \notin \mathcal{B}$
Output: $\pm(P \oplus Q) \in \mathcal{K}_{\mathcal{C}}^{\text{fast}}$.
Cost: $7\mathtt{M} + 8\mathtt{S} + 4\mathtt{m_c} + 24\mathtt{a}$

1 $(\mathsf{V}_1, \mathsf{V}_2) \leftarrow (\mathcal{H}(\mathcal{S}(\pm P)), \mathcal{H}(\mathcal{S}(\pm Q)))$ // 8S+16a
2 $\mathsf{V}_3 \leftarrow \mathcal{M}(\mathsf{V}_1, \mathsf{V}_2)$ // 4M
3 $\mathsf{V}_4 \leftarrow \mathcal{H}(\mathcal{M}(\mathsf{V}_3, (1/A : 1/B, 1/C, 1/D))$ // 4m_c+8a
4 **return** $\mathcal{M}(\mathsf{V}_4, (1 : x_\ominus/y_\ominus, x_\ominus/z_\ominus, x_\ominus/t_\ominus))$ // 3M

Algorithm 10: `xDBLADD`: Combined differential double-and-add on $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$.

Input: $(\pm P, \pm Q, (x_\ominus/y_\ominus, x_\ominus/z_\ominus, x_\ominus/t_\ominus) = \mathtt{Wrap}(\pm(P \ominus Q)))$ for P, Q in $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q)$ with $P \ominus Q \notin \mathcal{B}$.
Output: $(\pm[2]P, \pm(P \oplus Q))$
Cost: $7\mathtt{M} + 12\mathtt{S} + 12\mathtt{m_c} + 32\mathtt{a}$

1 $(\mathsf{V}_1, \mathsf{V}_2) \leftarrow (\mathcal{H}(\mathcal{S}(\pm P)), \mathcal{H}(\mathcal{S}(\pm Q)))$ // $8\mathtt{S} + 16\mathtt{a}$
2 $(\mathsf{V}_1, \mathsf{V}_2) \leftarrow (\mathcal{S}(\mathsf{V}_1), \mathcal{M}(\mathsf{V}_1, \mathsf{V}_2))$ // $4\mathtt{M} + 4\mathtt{S}$
3 $(\mathsf{V}_1, \mathsf{V}_2) \leftarrow \big(\mathcal{M}(\mathsf{V}_1, (\frac{1}{A} : \frac{1}{B} : \frac{1}{C} : \frac{1}{D})), \mathcal{M}(\mathsf{V}_2, (\frac{1}{A} : \frac{1}{B} : \frac{1}{C} : \frac{1}{D}))\big)$ // $8\mathtt{m_c}$
4 $(\mathsf{V}_1, \mathsf{V}_2) \leftarrow (\mathcal{H}(\mathsf{V}_1), \mathcal{H}(\mathsf{V}_2))$ // 16a
5 **return** $(\mathcal{M}(\mathsf{V}_1, (\frac{1}{a} : \frac{1}{b} : \frac{1}{c} : \frac{1}{d})), \mathcal{M}(\mathsf{V}_2, (1 : \frac{x_\ominus}{y_\ominus} : \frac{x_\ominus}{y_\ominus} : \frac{x_\ominus}{t_\ominus})))$ // $3\mathtt{M} + 4\mathtt{m_c}$

B Constant-time conditional swaps and selects

Our algorithms are designed to be a basis for uniform and constant-time implementations. As such, to avoid branching, we require constant-time conditional swap and selection routines. These are standard techniques, and can be implemented in many ways; Algorithms 11 and 12 give example pseudocode as an illustration of these techniques.

Algorithm 11: `SWAP`: Constant-time conditional swap.

Input: $b \in \{0,1\}$ and a pair (X_0, X_1) of objects encoded as n-bit strings
Output: (X_b, X_{1-b})

```
1 b ← (b, ..., b)_n
2 V ← b and (X_0 xor X_1)    // bitwise and, xor; do not short-circuit and
3 return (X_0 xor V, X_1 xor V)
```

Algorithm 12: `SELECT`: Constant-time conditional selection.

Input: $b \in \{0,1\}$ and a pair (X_0, X_1) of objects encoded as n-bit strings
Output: X_b

```
1 b ← (b, ..., b)_n
2 V ← b and (X_0 xor X_1)    // bitwise and, xor; do not short-circuit and
3 return X_0 xor V
```