Fast, uniform, and compact scalar multiplication for elliptic curves and genus 2 Jacobians with applications to signature schemes

Ping Ngai Chung, Craig Costello, Benjamin Smith

Fast, uniform, and compact scalar multiplication for elliptic curves and genus 2 Jacobians with applications to signature schemes

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), France
smith@lix.polytechnique.fr

Abstract. We give a general framework for uniform, constant-time one- and two-dimensional scalar multiplication algorithms for elliptic curves and Jacobians of genus 2 curves that operate by projecting to the x-line or Kummer surface, where we can exploit faster and more uniform pseudomultiplication, before recovering the proper "signed" output back on the curve or 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. Our results show that many existing fast pseudomultiplication implementations (hitherto limited to applications in Diffie–Hellman key exchange) can be wrapped with simple and efficient pre- and post-computations to yield competitive full scalar multiplication algorithms, ready for use in more general discrete logarithm-based cryptosystems, including signature schemes. This is especially interesting for genus 2, where Kummer surfaces can outperform comparable elliptic curve systems. As an example, we construct an instance of the Schnorr signature scheme driven by Kummer surface arithmetic.

1 Introduction

In terms of per-bit security, elliptic curves and Jacobians of genus 2 curves appear to be roughly equivalent. However, when it comes to efficient and side-channel-aware implementations, we see a curious divergence. Full genus 2 Jacobian arithmetic is relatively slow compared with elliptic curves, and hard to implement in a uniform and constant-time fashion. On the other hand, we have an extremely fast and uniform scalar pseudomultiplication for genus 2 Kummer surfaces, which often outperforms its elliptic equivalent; but this comes at the cost of identifying elements with their inverses, so Kummer surfaces are widely believed to be suitable only for Diffie–Hellman protocols, where no individual full group operations are required.

Thus, genus 2 beats elliptic curve performance for Diffie–Hellman (where we can use Kummer surfaces), but is beaten in almost every other implementation

scenario (where we are confined to Jacobians). In this article we show how to exploit Kummer surface arithmetic to carry out full Jacobian scalar multiplications, bringing the speed and side-channel security of Kummers to implementations of other discrete-log-based cryptographic protocols. In particular, our results show that many existing competitive Diffie–Hellman implementations based on pseudomultiplication can be wrapped with simple and efficient pre- and post-computations to yield competitive full scalar multiplication algorithms, ready for use in more general cryptosystems, including signature schemes.

Scalar multiplication. To make things precise, let $\mathcal{G}$ be a subgroup of an elliptic curve or a genus 2 Jacobian (with $\oplus$ denoting the group law, and $\ominus$ its inverse). We are interested in computing one-dimensional scalar multiplications

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

which is the operation at the heart of all discrete logarithm and Diffie–Hellman problem-based cryptosystems. We are also interested in two-dimensional multi-scalar multiplications

$$((m, n), (P, Q)) \longmapsto [m]P \oplus [n]Q \quad \text{for} \quad m, n \in \mathbb{Z}_{\geq 0}, P, Q \in \mathcal{G} \ ;$$

these appear explicitly in many cryptographic protocols, including signature verification, but they are also a key ingredient in endomorphism-accelerated one-dimensional scalar multiplication techniques such as GLV [18] and its descendants.

If the scalar m is secret, then $[m]P$ must be computed in 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).

Uniform, constant-time algorithms for elliptic curve scalar multiplication are well-known (and even widely-used). In contrast, if $\mathcal{G}$ is a subgroup of a genus 2 Jacobian, then this requirement has represented an insurmountable obstacle until now: the usual Cantor arithmetic [10] and its derivatives [23] have so many incompatible special cases that it has appeared impossible to implement it in a uniform way without abandoning all hope of competitive efficiency (see §6).

Kummer surfaces and x-lines. The situation changes dramatically when we pass to the quotient $\mathcal{G}/\langle \pm 1 \rangle$, identifying elements with their inverses. Let

$$x : \mathcal{G} \longrightarrow \mathcal{G}/\langle \pm 1 \rangle$$

be the quotient map, so $x(P) = x(\ominus P)$ for all P in $\mathcal{G}$. In the elliptic case, x is projection onto the x-coordinate (whence our notation); in genus 2, x is the map from the Jacobian to its Kummer surface. We emphasize that $\mathcal{G}/\langle \pm 1 \rangle$ is *not* a group, and at first glance this prevents us instantiating group-based

protocols in $\mathcal{G}/\langle\pm 1\rangle$. Nevertheless, scalar multiplication in $\mathcal{G}$ induces a well-defined *pseudomultiplication*

$$(m, x(P)) \longmapsto x([m]P)$$

in $\mathcal{G}/\langle\pm 1\rangle$, because $[m](\pm P) = \pm([m]P)$. This suffices for implementing protocols like Diffie–Hellman key exchange which *only* involve scalar multiplication, and not individual group operations (which we lose in passing from $\mathcal{G}$ to $\mathcal{G}/\langle\pm 1\rangle$).

Pseudomultiplication algorithms are typically faster and simpler than full scalar multiplication algorithms—the x-only Montgomery ladder being a case in point for elliptic curves—and these algorithms have therefore become the basis of the fastest and safest Diffie–Hellman implementations (such as Bernstein's Curve25519 software [1] and its heirs). We also see excellent Diffie–Hellman implementations based on Kummer surfaces [4].

But it is widely believed that pseudomultiplication cannot be used for general cryptosystems, because $\mathcal{G}/\langle\pm 1\rangle$ is not a group: since the "sign" of the output of a pseudomultiplication is ambiguous, it cannot be used as the input to individual group operations. This belief has so far disqualified Kummers as candidates for implementing most common signature schemes, as well as encryption schemes such as ElGamal [17]. As a result, we tend not to see highly competitive genus 2 implementations of signatures and public-key encryption schemes, because we are hamstrung by relatively slow and non-uniform Jacobian arithmetic.

Recovering group elements after pseudomultiplication. The folklore that Kummer surfaces cannot be used in true group-based cryptosystems may appear mathematically true—but it is algorithmically false.

Indeed, it has long been known that x-only arithmetic on elliptic curves can be used for full scalar multiplication: López and Dahab [27] (followed by Okeya and Sakurai [31] and Brier and Joye [8]) 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.

In this work we extend this technique from elliptic curves to genus 2, and from one- to two-dimensional scalar multiplication. In the abstract, our algorithms follow the same common pattern:

1. First, `Project` the inputs (and possibly some auxiliary elements) from $\mathcal{G}$ to $\mathcal{G}/\langle\pm 1\rangle$ using the x map;
2. then pseudomultiply in $\mathcal{G}/\langle\pm 1\rangle$ (that is, compute $x([m]P)$ or $x([m]P \oplus [n]Q)$ using a differential addition chain);
3. finally, efficiently `Recover` the correct preimage $[m]P$ (or $[m]P \oplus [n]Q$) in $\mathcal{G}$ from the outputs of the pseudomultiplication.

The one-dimensional version of this pattern in §3 uses the well-known Montgomery ladder [29] for its pseudomultiplication; the two-dimensional version in §4 is based on Bernstein's binary differential addition chain [2].

In §5 we apply our algorithms to various models of elliptic curves. We recover Okeya–Sakurai and Brier–Joye multiplication, along with new uniform, compact two-dimensional scalar multiplication algorithms.

Moving to genus 2, the Jacobian point recovery method we present in §6 solves the problem of uniform genus 2 arithmetic (at least for scalar multiplication): rather than wrestling with the special cases of Cantor's algorithm, we can descend to the faster, more uniform Kummer surface, pseudomultiply there, and then recover the right Jacobian point afterwards. Our methods work on the most general form of the Kummer, and are easy to adapt to more specialized models.

In §7 we specialize to the Gaudry model for Kummer surfaces, which have the fastest known arithmetic. The result is exceptionally fast one- and two-dimensional scalar multiplication algorithms for the genus 2 Jacobians that admit these Kummer surfaces. Finally, in §8 we give a concrete example of how our algorithms can be applied to create fast instances of Schnorr signature schemes.

Remark 1. Damien Robert has pointed out to us that his recent preprint with David Lubicz [28] uses similar techniques to improve the efficiency of their explicit arithmetic of general abelian varieties in arbitrarily high dimension based on theta functions. See Remark 6 below for further details.

Notation and conventions Throughout, $\mathbb{F}_q$ denotes a finite field of characteristic > 3. As usual, we use $\mathbf{M}$, $\mathbf{S}$, $\mathbf{I}$ and $\mathbf{a}$ to respectively denote the costs of one field multiplication, squaring, inversion, and addition in $\mathbb{F}_q$. We also use $\mathbf{m}_c$ to denote the cost of a field multiplication by a fixed constant c, which is often significantly different to $\mathbf{M}$, i.e., when c is a *small* curve constant.

In our algorithms, we use the notation $(x_1, \ldots, x_n) \leftarrow (y_1, \ldots, y_n)$ to denote *parallel assignment*. In sequential terms, $(x, y) \leftarrow (z, w)$ is equivalent to $x \leftarrow z; y \leftarrow w$ (and to $y \leftarrow w; x \leftarrow z$), while $(x, y) \leftarrow (y, x)$, which swaps x and y, is equivalent to $t \leftarrow x; x \leftarrow y; y \leftarrow t$, where t is a temporary variable.

2 Key subroutines

In this paper we work with various models of elliptic curves, Jacobians, and Kummer surfaces, but viewed from a high level the algorithms are essentially the same; we can therefore make some substantial simplifications by presenting them as template algorithms acting on an abstract abelian group $\mathcal{G}$, and its quotient $\mathcal{G}/\pm 1$, by black-box subroutines (whose implementation details we will provide later). We therefore assume the existence of six algorithms acting on elements of $\mathcal{G}$ and $\mathcal{G}/\langle\pm 1\rangle$:

1. $\underline{\texttt{Project} : \mathcal{G} \to \mathcal{G}/\langle\pm 1\rangle}$ implements the mapping $x : \mathcal{G} \to \mathcal{G}/\langle\pm 1\rangle$; that is,

$$\texttt{Project}(P) = x(P) \quad \text{for all } P \in \mathcal{G} \ .$$

   This is trivial in the elliptic context, where it amounts to dropping one of the coordinates, and only slightly less straightforward in genus 2.

2. $\underline{\texttt{xDBL} : \mathcal{G}/\langle\pm1\rangle \to \mathcal{G}/\langle\pm1\rangle}$ implements pseudo-doubling in $\mathcal{G}/\langle\pm1\rangle$: that is,

$$\texttt{xDBL}(x(P)) = x([2]P) \quad \text{for all } P \in \mathcal{G} .$$

Efficient formulæ for `xDBL` are known in each of our contexts.

3. $\underline{\texttt{xADD} : (\mathcal{G}/\langle\pm1\rangle)^3 \to \mathcal{G}/\langle\pm1\rangle}$ implements the standard differential addition:

$$\texttt{xADD}(x(P), x(Q), x(P \ominus Q)) = x(P \oplus Q) \quad \text{for all } P \neq Q \in \mathcal{G} .$$

As with `xDBL`, efficient formulæ for `xADD` are known in each of our contexts.

4. $\underline{\texttt{xDBLADD} : (\mathcal{G}/\langle\pm1\rangle)^3 \to (\mathcal{G}/\langle\pm1\rangle)^2}$ implements a simultaneous doubling and differential addition:

$$\texttt{xDBLADD}(x(P), x(Q), x(Q \ominus P)) = (x([2]P), x(P \oplus Q)) \quad \text{for all } P \neq Q \in \mathcal{G} .$$

While `xDBLADD` may be naïvely defined by computing an `xDBL` and an `xADD` separately—that is, as

$$\texttt{xDBLADD}(x(P), x(Q), x(Q \ominus P)) = (\texttt{xDBL}(x(P)), \texttt{xADD}(x(P), x(Q), x(Q \ominus P)))$$

—sometimes in practice it can be implemented more efficiently by exploiting shared intermediate operands, so we treat it as a distinct operation.

5. $\underline{\texttt{ADD} : \mathcal{G} \times \mathcal{G} \to \mathcal{G}}$ computes the group law in $\mathcal{G}$:

$$\texttt{ADD}(P, Q) = P \oplus Q \quad \text{for all } P, Q \in \mathcal{G} .$$

`ADD` is only used once in the two-dimensional algorithm; it is not used at all in the one-dimensional algorithm. Minimizing `ADD`s is a deliberate strategy: in practice, it is often a relatively costly and potentially non-constant-time operation compared with `xADD` (especially in genus 2).

6. $\underline{\texttt{Recover} : \mathcal{G} \times (\mathcal{G}/\langle\pm1\rangle)^2 \to \mathcal{G}/\langle\pm1\rangle}$ computes preimages under x:

$$\texttt{Recover}(P, x(Q), x(Q \oplus P)) = Q \quad \text{for all } P, Q \in \mathcal{G} \setminus \mathcal{G}[2] .$$

This operation was introduced for binary elliptic curves by López and Dahab [27], for Montgomery models of elliptic curves by Okeya and Sakurai [31], and for short Weierstrass models of elliptic curves by Brier and Joye [8]. We extend these techniques to genus 2 in §6 and §7.

3 Uniform one-dimensional scalar multiplication

Algorithm 1 is a template for uniform one-dimensional scalar multiplication; it is suitable for use anywhere in curve-based cryptosystems where the calculation $(m, P) \mapsto [m]P$ is required, but especially those where m is secret. Algorithm 1 applies the `Project`-pseudomultiply-`Recover` pattern to lift the x-only Montgomery ladder for pseudomultiplication in $\mathcal{G}/\langle\pm1\rangle$ to a full scalar multiplication routine for $\mathcal{G}$, generalizing and abstracting the methods of [27], [31], and [8].

Input : A positive integer $m = \sum_{i=0}^{\beta-1} m_i 2^i$, with $m_{\beta-1} \neq 0$, and an element P of $\mathcal{G}$
Output: $R = [m]P$

```
1  x_P ← Project(P) ;
2  (t_1, t_2) ← (x_P, xDBL(x_P)) ;
3  for i = β − 2 down to 0 do
4  |   if m_i = 0 then
5  |   |   (t_1, t_2) ← xDBLADD(t_1, t_2, x_P) ;
6  |   else
7  |   |   (t_2, t_1) ← xDBLADD(t_2, t_1, x_P) ;
8  |   end
9  end
10 R ← Recover(P, t_1, t_2) ;
11 return R ;
```

Algorithm 1: A one-dimensional uniform scalar multiplication template, based on the Montgomery ladder

Lemma 1. *Let m be a non-negative integer of bitlength β, and P an element of $\mathcal{G}$. Algorithm 1 computes $[m]P$ using one call to* `Project`*, one call to* `xDBL`*, $\beta - 1$ calls to* `xDBLADD`*, and one call to* `Recover`*.*

Proof. Line 1 of Algorithm 1 calls `Project` to map P into $\mathcal{G}/\langle\pm 1\rangle$. Lines 2-9 are just the standard Montgomery ladder [29]. At the end of each of the β iterations of the loop (each of which calls `xDBLADD` once), we have

$$(t_1, t_2) = (x([\lfloor m/2^i \rfloor]P), x([\lfloor m/2^i \rfloor + 1]P)) ;$$

so at the end of the loop, at Line 10, $(t_1, t_2) = (x([m]P), x([m]P \oplus P))$. The `Recover`$(P, t_1, t_2)$ in Line 10 therefore yields $[m]P$. □

In its abstract form, Algorithm 1 is uniform and constant-time with respect to fixed-length m. In practice, the implementation of `xDBLADD` must also be uniform and constant-time. If uniform, constant-time behaviour is required with respect to P, then the implementation of `Project` must also be uniform and constant-time.

4 Uniform two-dimensional scalar multiplication

Algorithm 3 is a template for uniform two-dimensional scalar multiplication. It is intended for use in cryptographic routines that require computing $[m]P \oplus [b]Q$, especially when at least one of m and n are secret, but it is also useful for implementing endomorphism-accelerated scalar multiplications, which compute $[m]P$ as $[m_0]P \oplus [m_1]\phi(P)$. Algorithm 3 applies the `Project`-pseudomultiply-`Recover` pattern to a pseudomultiplication based on Bernstein's binary differential addition chain [2]. Our algorithm is similar to its x-only counterpart, which was used (without any proof of correctness) in [14].

4.1 Bernstein's binary differential addition chain

Bernstein defined his binary differential addition chain in [2, §4] as follows. First, set

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

then the chain $C_D(A,B)$ is defined recursively by

$$C_D(A,B) := C_d(a,b) \,||\, (O,E,M)$$

where $||$ denotes concatenation, O, E, and M are defined by

$$O := (A + (A+1 \bmod 2), B + (B+1 \bmod 2)) \ , \tag{1}$$

$$E := (A + (A+0 \bmod 2), B + (B+0 \bmod 2)) \ , \tag{2}$$

$$M := (A + (A+D \bmod 2), B + (B+D+1 \bmod 2)) \ , \tag{3}$$

and a, b, and d are defined by

$$a := \lfloor A/2 \rfloor \ , \quad b := \lfloor B/2 \rfloor \ , \quad d := \begin{cases} D & \text{if } a \equiv A \text{ and } b \equiv B \pmod 2 \ , \\ 0 & \text{if } a \equiv A \text{ and } b \not\equiv B \pmod 2 \ , \\ 1 & \text{if } a \not\equiv A \text{ and } b \equiv B \pmod 2 \ , \\ 1-D & \text{if } a \not\equiv A \text{ and } b \not\equiv B \pmod 2 \ . \end{cases}$$

Observe that O contains two odd integers, E two even integers, and M is "mixed", with one even and one odd integer. The differences $M-O$, $M-E$, and $O-E$ depend only on D and the parities of A and B, as shown in Table 1.

Table 1. 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	$(-1,-1)$	$(1-D,D)$	$(-D,D-1)$

By definition, the triple (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 $(A + (A+D+1 \bmod 2), B + (B+D \bmod 2))$, from which it follows immediately that

$$C_{(A \bmod 2)}(A,B) \text{ contains } (A,B) \ .$$

Lemma 2. *With the notation above: Suppose $(a,b) \neq (0,0)$, and write o,e,m for the last three terms of $C_d(a,b)$, so $C_D(A,B) = (\ldots,o,e,m,O,E,M)$. Then O, E, and M can be expressed in terms of o, e, m, D, and the parities of A, B, a, and b using the relations in Table 2.*

Proof. The result follows—after elementary but lengthy calculations—from the definition of $C_D(A,B)$, considerations of parity, and the values in Table 1 applied to o, e, and m, with d derived from the first four columns. □

Table 2. Relations between the adjacent triples (o, e, m) and (O, E, M).

$A-B$ (mod 2)	$A-a$ (mod 2)	$B-b$ (mod 2)	D	O	difference of summands	E	M	difference of summands
0	0	0	0	$o+e$	$\pm(1,1)$	$2e$	$m+e$	$\pm(0,1)$
0	0	0	1	$o+e$	$\pm(1,1)$	$2e$	$m+e$	$\pm(1,0)$
0	0	1	0	$o+e$	$\pm(1,1)$	$2m$	$m+e$	$\pm(0,1)$
0	0	1	1	$o+e$	$\pm(1,1)$	$2m$	$m+o$	$\pm(1,0)$
0	1	0	0	$o+e$	$\pm(1,1)$	$2m$	$m+o$	$\pm(0,1)$
0	1	0	1	$o+e$	$\pm(1,1)$	$2m$	$m+e$	$\pm(1,0)$
0	1	1	0	$o+e$	$\pm(1,1)$	$2o$	$m+o$	$\pm(0,1)$
0	1	1	1	$o+e$	$\pm(1,1)$	$2o$	$m+o$	$\pm(1,0)$
1	0	0	0	$o+e$	$\pm(1,-1)$	$2e$	$m+e$	$\pm(0,1)$
1	0	0	1	$o+e$	$\pm(1,-1)$	$2e$	$m+e$	$\pm(1,0)$
1	0	1	0	$o+e$	$\pm(1,-1)$	$2m$	$m+e$	$\pm(0,1)$
1	0	1	1	$o+e$	$\pm(1,-1)$	$2m$	$m+o$	$\pm(1,0)$
1	1	0	0	$o+e$	$\pm(1,-1)$	$2m$	$m+o$	$\pm(0,1)$
1	1	0	1	$o+e$	$\pm(1,-1)$	$2m$	$m+e$	$\pm(1,0)$
1	1	1	0	$o+e$	$\pm(1,-1)$	$2o$	$m+o$	$\pm(0,1)$
1	1	1	1	$o+e$	$\pm(1,-1)$	$2o$	$m+o$	$\pm(1,0)$

4.2 Encoding the chain

Our aim is to use $C_{m_0}(m,n)$ to compute $x([m]P \oplus [n]Q)$, where

$$m = \sum_{i=0}^{\beta-1} m_i 2^i \quad \text{and} \quad n = \sum_{i=0}^{\beta-1} n_i 2^i \ .$$

Treating the m_i and n_i as bits, with $\oplus$ denoting binary addition (xor) and $\otimes$ binary multiplication (and), we define the sequence of *transition vectors*

$$c_i := ((m_i \oplus n_i), (m_i \oplus m_{i+1}), (n_i \oplus n_{i+1}), d_i) \ \text{ for } 0 \le i < \beta - 1 \ ,$$

where $d_0 := m_0$ and

$$d_{i+1} := ((d_i \oplus 1) \otimes (m_i \oplus m_{i+1})) \oplus (d_i \otimes (n_i \oplus n_{i+1} \oplus 1)) \text{ for } i \ge 0 \ .$$

The coordinates of c_i correspond to the first four columns of Table 2, with $A = \lfloor m/2^i \rfloor$, $a = \lfloor m/2^{(i+1)} \rfloor$, $B = \lfloor n/2^i \rfloor$, $b = \lfloor n/2^{(i+1)} \rfloor$, $D = d_i$, and $d = d_{i+1}$.

In view of Lemma 2, given m and n, we can iteratively reconstruct the entire chain $C_{m_0}(m,n)$ from $d_{\beta-1}$ and the sequence of transition vectors $c_0, \ldots, c_{\beta-2}$. Algorithm 2 computes precisely this data to encode $C_{m_0}(m,n)$. We note that the c_i also encode the difference elements required for each differential addition.

4.3 Two-dimensional scalar multiplication

If we map each pair (a,b) in $C_{m_0}(m,n)$ to the element $x([a]P \oplus [b]Q)$ in $\mathcal{G}/\langle \pm 1 \rangle$, then $C_{m_0}(m,n)$ yields a method of computing $x([m]P \oplus [n]Q)$ using a sequence of

Input : Positive β-bit integers $m = \sum_{i=0}^{\beta-1} m_i 2^i$ and $n = \sum_{i=0}^{\beta-1} n_i 2^i$.
Output: A sequence C of $\beta - 1$ transition vectors and one additional bit $d_{\beta-1}$, encoding the differential addition chain $\mathcal{C}_{m_0}(m, n)$.

1 $C \leftarrow [\,]$;
2 $d \leftarrow m_0$;
3 **for** $i \leftarrow 0$ *up to* $\beta - 2$ **do**
4 | Append $((m_i \oplus m_{i+1}), (n_i \oplus n_{i+1}), (m_{i+1} \oplus n_{i+1}), d)$ to C ;
5 | $d \leftarrow ((d \oplus 1) \otimes (m_i \oplus m_{i+1})) \oplus (d \otimes (n_i \oplus n_{i+1} \oplus 1))$;
6 **end**
7 **return** (C, d) ;

Algorithm 2: `ChainEncoding`: Computes and encodes Bernstein's two-dimensional "binary" differential addition chain

`xADD`s (and `xDBL`s) with the fixed differences $x(P)$, $x(Q)$, $x(P \oplus Q)$, and $x(P \ominus Q)$. Thus, the sequence of transition vectors in our encoding of $C_{m_0}(m, n)$ gives us a natural iterative algorithm for computing $x([m]P \oplus [n]Q)$ starting from the fixed differences. Using this as the core of our `Project`-pseudomultiply-`Recover` pattern yields Algorithm 3, which computes $[m]P \oplus [n]Q$ in $\mathcal{G}$.

Theorem 1. *Let P and Q be elements of $\mathcal{G}$, let m and n be positive integers, and let β be the bitlength of* $\max(m, n)$*. Algorithm 3 computes $[m]P \oplus [n]Q$ using one call to* `ADD`*, three calls to* `Project`*, β calls to each of* `xADD` *and* `xDBLADD`*, and one call to* `Recover`*.*

Proof. Algorithm 3 begins by calling Algorithm 2 to compute the sequence of transition vectors and $d_{\beta-1}$ for the given m and n. Lines 2–4 compute the required differences $x_P := x(P)$, $x_Q := x(Q)$, $x_\oplus := x(P \oplus Q)$, and $x_\ominus := x(P \ominus Q)$ by computing $S := P \oplus Q$ with the single call to `ADD`, then using the three calls to `Project` to compute x_P, x_Q, and $x_\oplus = x(S)$, before $x_\ominus = \texttt{xADD}(x_P, x_Q, x_\oplus)$.

Throughout the rest of the algorithm O corresponds to the odd-odd pair, E to the even-even pair, and M to the mixed pair in a segment of $C_{m_0}(m, n)$, where each pair (a, b) is associated with $x([a]P \oplus [b]Q)$. Lines 5–12 use a single `xDBLADD` to initialize O, E, and M such that $C_{d_{\beta-1}}(m_{\beta-1}, n_{\beta-1}) = C_0(0, 0)||(O, E, M)$. Lines 13–31 iterate over the sequence of transition vectors to compute the last three elements of $C_{m_0}(m, n)$. After each iteration, the triple (O, E, M) satisfies

$$C_{d_i}(\lfloor m/2^i \rfloor, \lfloor n/2^i \rfloor) = C_{d_{i+1}}(\lfloor m/2^{i+1} \rfloor, \lfloor n/2^{i+1} \rfloor) \;||\; (O, E, M) \;.$$

Each iteration requires two differential additions and a pseudodoubling, but we observe that the pseudodoubling always applies to one of the arguments of a differential addition, so each of the $\beta - 1$ iterations requires exactly one `xADD` and one `xDBLADD`.

After the loop is completed, at Line 33, Eqs. (1), (2), and (3) imply

$$O = x(R \oplus \Delta_O) \;, \quad E = x(R \oplus \Delta_E) \;, \quad \text{and} \quad M = x(R \oplus \Delta_M) \;,$$

where $R = [m]P \oplus [n]Q$ and Δ_O, Δ_E, and Δ_M are given in the following table:

Input : Positive β-bit integers $m = \sum_{i=0}^{\beta-1} m_i 2^i$ and $n = \sum_{i=0}^{\beta-1} n_i 2^i$ with $m_{\beta-1}$ and $n_{\beta-1}$ not both zero, and elements P, Q of $\mathcal{G}$
Output: $R = [m]P \oplus [n]Q$

$((c_0, \ldots, c_{\beta-2}), d_{\beta-1}) \leftarrow \texttt{ChainEncoding}(m,n)$;
$S \leftarrow \texttt{ADD}(P, Q)$;
$(x_P, x_Q, x_\oplus) \leftarrow (\texttt{Project}(P), \texttt{Project}(Q), \texttt{Project}(S))$;
$x_\ominus \leftarrow \texttt{xADD}(x_P, x_Q, x_\oplus)$;
switch $(m_{\beta-1}, n_{\beta-1}, d_{\beta-1})$ **do**
  case $(0,1,0)$: $(\mathsf{M}, (\mathsf{E}, \mathsf{O})) \leftarrow (x_Q, \texttt{xDBLADD}(x_Q, x_P, x_\ominus))$;
  case $(0,1,1)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (x_\oplus, \texttt{xDBLADD}(x_Q, x_\oplus, x_P))$;
  case $(1,0,0)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (x_\oplus, \texttt{xDBLADD}(x_P, x_\oplus, x_Q))$;
  case $(1,0,1)$: $(\mathsf{M}, (\mathsf{E}, \mathsf{O})) \leftarrow (x_P, \texttt{xDBLADD}(x_P, x_Q, x_\ominus))$;
  case $(1,1,0)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (x_\oplus, \texttt{xDBLADD}(x_\oplus, x_P, x_Q))$;
  case $(1,1,1)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (x_\oplus, \texttt{xDBLADD}(x_\oplus, x_Q, x_P))$;
endsw
for $i \leftarrow \beta - 2$ *down to* 0 **do**
  switch c_i **do**
    case $(0,0,0,0)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\oplus), \texttt{xDBLADD}(\mathsf{E}, \mathsf{M}, x_Q))$;
    case $(0,0,0,1)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\oplus), \texttt{xDBLADD}(\mathsf{E}, \mathsf{M}, x_P))$;
    case $(0,0,1,0)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\oplus), \texttt{xDBLADD}(\mathsf{M}, \mathsf{E}, x_Q))$;
    case $(0,0,1,1)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\oplus), \texttt{xDBLADD}(\mathsf{M}, \mathsf{O}, x_P))$;
    case $(0,1,0,0)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\oplus), \texttt{xDBLADD}(\mathsf{M}, \mathsf{O}, x_Q))$;
    case $(0,1,0,1)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\oplus), \texttt{xDBLADD}(\mathsf{M}, \mathsf{E}, x_P))$;
    case $(0,1,1,0)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\oplus), \texttt{xDBLADD}(\mathsf{O}, \mathsf{M}, x_Q))$;
    case $(0,1,1,1)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\oplus), \texttt{xDBLADD}(\mathsf{O}, \mathsf{M}, x_P))$;
    case $(1,0,0,0)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\ominus), \texttt{xDBLADD}(\mathsf{E}, \mathsf{M}, x_Q))$;
    case $(1,0,0,1)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\ominus), \texttt{xDBLADD}(\mathsf{E}, \mathsf{M}, x_P))$;
    case $(1,0,1,0)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\ominus), \texttt{xDBLADD}(\mathsf{M}, \mathsf{E}, x_Q))$;
    case $(1,0,1,1)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\ominus), \texttt{xDBLADD}(\mathsf{M}, \mathsf{O}, x_P))$;
    case $(1,1,0,0)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\ominus), \texttt{xDBLADD}(\mathsf{M}, \mathsf{O}, x_Q))$;
    case $(1,1,0,1)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\ominus), \texttt{xDBLADD}(\mathsf{M}, \mathsf{E}, x_P))$;
    case $(1,1,1,0)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\ominus), \texttt{xDBLADD}(\mathsf{O}, \mathsf{M}, x_Q))$;
    case $(1,1,1,1)$: $(\mathsf{O}, (\mathsf{E}, \mathsf{M})) \leftarrow (\texttt{xADD}(\mathsf{O}, \mathsf{E}, x_\ominus), \texttt{xDBLADD}(\mathsf{O}, \mathsf{M}, x_P))$;
  endsw
end
switch (m_0, n_0) **do**
  case $(0,0)$: $R \leftarrow \texttt{Recover}(S, \mathsf{E}, \mathsf{O})$;
  case $(0,1)$: $R \leftarrow \texttt{Recover}(P, \mathsf{M}, \mathsf{O})$;
  case $(1,0)$: $R \leftarrow \texttt{Recover}(P, \mathsf{M}, \mathsf{E})$;
  case $(1,1)$: $R \leftarrow \texttt{Recover}(S, \mathsf{O}, \mathsf{E})$;
endsw
return R ;

Algorithm 3: Two-dimensional uniform scalar multiplication template

(m_0, n_0)	Δ_O	Δ_E	Δ_M if $d_0 = 0$	Δ_M if $d_0 = 1$	$[m]P \oplus [n]Q$
$(0,0)$	$P \oplus Q$	$0_{\mathcal{G}}$	Q	P	$\texttt{Recover}(P \oplus Q, E, O)$
$(0,1)$	P	Q	$0_{\mathcal{G}}$	$P \oplus Q$	$\texttt{Recover}(P, M, O)$
$(1,0)$	Q	P	$P \oplus Q$	$0_{\mathcal{G}}$	$\texttt{Recover}(P, M, E)$
$(1,1)$	$0_{\mathcal{G}}$	$P \oplus Q$	P	$P \oplus Q$	$\texttt{Recover}(P \oplus Q, O, E)$

In each case, we can recover $[m]P \oplus [n]Q$ by applying `Recover` with the arguments specified by the last column of the corresponding row. This is precisely what is done in Lines 33-38; Line 39 then returns the result, $[m]P \oplus [n]Q$. □

Like Algorithm 1, Algorithm 3 is uniform and constant-time in its abstract form, at least for fixed-length multiscalars (m, n). In practice, for Algorithm 3 to be uniform and constant-time with respect to m and n, the implementations of `xADD` and `xDBLADD` must be uniform and constant-time. For uniform and constant-time behaviour with respect to P and Q, the implementations of `ADD` and `Project` must also be uniform and constant-time.

Remark 2. The core of Algorithm 3 is similar to the algorithm in [14, App. C], but with a different (and slightly simpler) encoding of the addition chain. Here, the i-th transition vector is $((m_i \oplus n_i), (m_i \oplus m_{i+1}), (n_i \oplus n_{i+1}), d_i)$; in [14, App. C] it is $((m_i \oplus m_{i+1} \oplus n_i \oplus n_{i+1}), (m_i \oplus m_{i+1}), (m_i \oplus n_i), d_i)$, and the order of the vectors is reversed (as is the order of the loop iterations).

Remark 3. Implementers should notice that every adjacent pair of `xADD` and `xDBLADD` operations in Algorithm 3 share one operand in their differential additions. Further savings might therefore be made by sharing a few intermediate calculations between the `xADD` and `xDBLADD` calls—that is, by implicitly defining an `xDBLADDADD` operation, as in [14, App. C]. We have not done this here, for two reasons. First, it somewhat obscures the fundamental symmetry of the addition chain. Second, looking ahead at the explicit formulæ for `xADD` and `xDBLADD` suggests that there are no intermediate calculations that can be shared in the elliptic curve scenarios. However, close inspection of the fast Kummer arithmetic in §7.3 reveals that two adjacent differential additions sharing a common summand can be merged to save 8 field additions. It is unclear to us that whether this potential saving would give a net benefit after the code complexity required to perform the merging, so we have not included this potential saving in our operation counts.

Remark 4. These techniques should readily extend to the higher-dimensional Montgomery-like differential addition chains described by Brown [9]. We do not investigate this here.

5 Efficient scalar multiplication on elliptic curves

We now pass from the abstract to the concrete. Applying the one-dimensional pattern to elliptic curves, we recover Okeya–Sakurai-style multiplication algorithms; applying the two-dimensional pattern yields something new.

5.1 Montgomery models

The most obvious application of our methods is to elliptic curves with Montgomery models

$$BY^2Z = X(X^2 + AXZ + Z^2) ,$$

which are defined to optimize the `xADD`, `xDBL`, and `xDBLADD` operations. Indeed, historically, the first appearance of the `Project`-pseudomultiply-`Recover` pattern was in the context of one-dimensional scalar multiplication on Montgomery models [31]. Important examples of Montgomery curves include Curve25519, recently recommended for standardization by the CFRG.

In this context, `Project` : $(x, y) \mapsto x$ is trivially computed. The `ADD`, `xADD`, `xDBL`, and `xDBLADD` operations are all presented—and thoroughly costed—in the EFD [6]. The `Recover` operation was derived by Okeya and Sakurai in [31, §3]. The operation counts for all of these operations are compiled in Table 3.

Table 3. Costs of operations for projective Montgomery models $By^2 = x(x^2+Ax+1)$, where $\mathbf{m}_A$ denotes multiplications by A and $(A+2)/4$, and $\mathbf{m}_B$ denotes multiplications by B. A point is normalized if its projective Z-coordinate has been scaled to 1.

Algorithm	**M**	**S**	$\mathbf{m}_A$	$\mathbf{m}_B$	**a**	**I**	Conditions
`ADD`	1	1	0	0	5	1	P and Q normalized
`Project`	0	0	0	0	0	0	—
`xDBL`	2	5	1	0	0	0	—
`xADD`	4	2	0	0	6	0	—
`xADD`	3	2	0	0	6	0	$P \ominus Q$ normalized
`xDBLADD`	6	4	1	0	8	0	—
`xDBLADD`	5	4	1	0	8	0	$P \ominus Q$ normalized
`Recover`	13	1	1	1	8	1	P normalized

We now apply the `Project`-pseudomultiply-`Recover` pattern using the routines above. In the one-dimensional case, we recover the Okeya–Sakurai algorithm [31].

Theorem 2. *Let $\mathcal{E}/\mathbb{F}_q$ be an elliptic curve in Montgomery form.*

1. *Let P be a point in $\mathcal{E}(\mathbb{F}_q) \setminus \mathcal{E}[2]$, and let m be a positive β-bit integer. Then Algorithm 1 computes $[m]P$ in*

$$(5\beta + 10)\mathbf{M} + (4\beta + 2)\mathbf{S} + (\beta + 1)\mathbf{m}_A + 1\mathbf{m}_B + (8\beta + 6)\mathbf{a} + 1\mathbf{I} .$$

2. *Let P and Q be points in $\mathcal{E}(\mathbb{F}_q) \setminus \mathcal{E}[2]$, and let m and n be positive β-bit integers. Then Algorithm 3 computes $[m]P \oplus [n]Q$ in*

$$(8\beta + 14)\mathbf{M} + (6\beta + 2)\mathbf{S} + (\beta + 1)\mathbf{m}_A + 1\mathbf{m}_B + (14\beta + 13)\mathbf{a} + 2\mathbf{I} .$$

5.2 Edwards models

As a byproduct of Theorem 2, we obtain new scalar multiplication algorithms for Edwards models [16]. Indeed, every twisted Edwards model can be transformed into a Montgomery model with a linear change of coordinates, so one option to perform scalar multiplications on Edwards curves is to pass to and from an associated Montgomery model, making use of the operations summarized in Table 3. Important examples of Edwards models in practice include Ed25519 [5] and Goldilocks [22].

Another option here is to exploit pseudomultiplication formulæ that are native to Edwards curves, as described by Gaudry and Lubicz [20, §6.2]. In this case, for Edwards curves $\mathcal{E}/\mathbb{F}_q\colon x^2 + y^2 = 1 + dx^2y^2$ with $d = r^2$ and $r \in \mathbb{F}_q$, we define $\texttt{Project} : (x, y) \mapsto y$ and use the formulæ in the EFD [6] to compute the pseudomultiplication via Algorithm 1.

The $\texttt{Recover}\colon (P, y(Q), y(Q \oplus P)) \mapsto Q = (x(Q), y(Q))$ operation is defined by rearranging the Edwards addition law between P and Q, to make $x(Q)$ the subject, i.e.,

$$x(Q) = \frac{y(Q \oplus P) - y(Q)y(P)}{x(P)\left(dy(P)y(Q)y(Q \oplus P) - 1\right)} \ .$$

5.3 Scalar multiplication on short Weierstrass models

Not every elliptic curve has a Montgomery or Edwards model; the most general form for an elliptic curve over $\mathbb{F}_q$ in characteristic greater than 3 is the short Weierstrass model

$$\mathcal{E} : y^2 = x^3 + Ax + B \subset \mathbb{P}^2 \ .$$

Important examples of curves commonly implemented as Weierstrass models include the NIST [30] and Brainpool curves [7].

Our key subroutines are implemented as follows. For brevity, we use affine coordinates here, but the resulting formulæ are easily projectivized: see Brier and Joye or the EFD. In this context, $\texttt{Project} : P = (x, y) \mapsto x$ is trivially computed. The `ADD`, `xADD`, and `xDBL`, and `xDBLADD` operations are all specified as efficient straight-line programs in the EFD [6]. Brier and Joye describe `Recover` for short Weierstrass models in [8, Prop. 3]: it maps $(P, x(Q), x(P \oplus Q))$ to $Q = (x(Q), y(Q))$, where

$$y(Q) = \frac{2B + (A + x(P)x(Q))(x(P) + x(Q)) - x(P \oplus Q)(x(P) - x(Q))^2}{2y(P)} \ .$$

Table 4 summarizes the operation counts for all of this routines.

Applying Algorithm 1 with these subroutines yields the scalar multiplication algorithm for Weierstrass models in [8]. Applying Algorithm 3 with these subroutines yields a new algorithm for two-dimensional scalar multiplication on Weierstrass curves.

Theorem 3. *Let $\mathcal{E}/\mathbb{F}_q$ be an elliptic curve in short Weierstrass form.*

Table 4. Costs of operations for projective short Weierstrass models $y^2 = x^3 + ax + b$, where $\mathbf{m}_b$ denotes multiplications by b, $2b$ and $4b$. A point is normalized if its projective Z-coordinate has been scaled to 1.

Algorithm	**M**	**S**	$\mathbf{m}_a$	$\mathbf{m}_b$	**a**	**I**	Conditions
`ADD`	1	1	0	0	4	1	P and Q normalized
`Project`	0	0	0	0	0	0	—
`xDBL`	2	5	1	2	8	0	—
`xADD`	7	2	1	1	6	0	—
`xADD`	6	2	1	1	4	0	$P \ominus Q$ normalized
`xDBLADD`	9	7	2	3	12	0	—
`xDBLADD`	8	7	2	3	12	0	$P \ominus Q$ normalized
`Recover`	11	2	1	1	7	1	P normalized

1. *Let P be a point in $\mathcal{E}(\mathbb{F}_q) \setminus \mathcal{E}[2]$, and let m be a positive β-bit integer. Then Algorithm 1 computes $[m]P$ in*

$$(8\beta + 5)\mathbf{M} + 7\beta\mathbf{S} + 2\beta\mathbf{m}_a + 3\beta\mathbf{m}_b + (12\beta + 3)\mathbf{a} + 1\mathbf{I} \ .$$

2. *Let P and Q be points in $\mathcal{E}(\mathbb{F}_q) \setminus \mathcal{E}[2]$, and let m and n be positive β-bit integers. Then Algorithm 3 computes $[n]P \oplus [n]Q$ in*

$$(14\beta + 12)\mathbf{M} + (9\beta + 3)\mathbf{S} + (3\beta + 1)\mathbf{m}_a + (4\beta + 1)\mathbf{m}_b + (16\beta + 11)\mathbf{a} + 2\mathbf{I} \ .$$

Proof. Take the values from Table 4 in Lemma 1 (for the first part) and Theorem 1 (for the second). □

6 Genus 2 Jacobians and General Kummer Surfaces

We now turn our focus to genus 2 cryptosystems, where $\mathcal{G}$ is (a subgroup of) the Jacobian $\mathcal{J}_\mathcal{C}$ of a genus 2 curve

$$\mathcal{C} : y^2 = f(x) = f_6x^6 + f_5x^5 + \cdots + f_1x + f_0 \quad \text{over } \mathbb{F}_q \ .$$

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 quotient $\mathcal{G}/\langle \pm 1 \rangle$ is (a subset of) a Kummer surface, which is a quartic surface in $\mathbb{P}^3$. There are two main ways to represent the Kummer: the "general" model $\mathcal{K}_\mathcal{C}^{\text{gen}}$ (see Cassels and Flynn [11, Ch. 3]) and the "fast" model $\mathcal{K}_\mathcal{C}^{\text{fast}}$ algorithmically developed by Chudnovsky and Chudnovsky [12], and introduced in cryptography by Gaudry [19].

Broadly speaking, the fast Kummer model corresponds to the Montgomery model for elliptic curves, while the general model corresponds to the Weierstrass model. Indeed, as in the elliptic curve situation, fast Kummers offer significant gains in performance and uniformity (indeed, they are at the heart of a number of record-breaking Diffie–Hellman implementations), but at the price of a lot of rational 2-torsion: hence, not every Kummer can be put in fast form.

We will define the `Project` and `Recover` operations for general Kummers; since any Kummer surface can be transformed into a general model over the ground field, these `Project` and `Recover` routines can easily be specialized to fast Kummers (or to any other interesting models of Kummer surfaces). We do not lose much in taking this approach compared with deriving a new, specialized `Project` and `Recover` for Jacobians with fast Kummers from scratch, because Algorithms 1 and 3 only call `Project` a few times, and `Recover` once. Scalar pseudomultiplication on general Kummers is relatively slow, and has held little interest for cryptographers thus far; we briefly discuss their arithmetic and pseudomultiplication in §6.5.

6.1 Genus 2 arithmetic and side-channel attacks

The group law on $\mathcal{J}_\mathcal{C}$ (and hence the `ADD` operation) is typically computed using Cantor's algorithm, specialized to genus 2. This style of arithmetic has a serious drawback in cryptographic contexts: it is highly susceptibile to simple side-channel attacks. Similar to the textbook addition for short Weierstrass models of elliptic curves, Cantor's algorithm in genus 2 treats several input cases differently, *branching* off into distinct explicit computations. Explicit formulæ that are derived for generic additions fail to compute correctly when one or both of the inputs are *special* points—that is, points in $\mathcal{J}_\mathcal{C}$ where $a(x) = x - \alpha$ or $a(x) = (x-\alpha)^2$. While such special points are sparse enough in $\mathcal{J}_\mathcal{C}$ that random scalar multiplications do not encounter them, they are plentiful enough that attackers could easily mount exceptional procedure attacks [24], forcing legitimate users into special cases and using timing variabilities to recover secret data.

The `Project`-pseudomultiply-`Recover` approach detailed in §6.2 addresses this problem. Like x-only (pseudo)scalar multiplication on elliptic curves, the explicit formulæ for doublings and differential additions on certain Kummer surfaces associated to $\mathcal{J}_\mathcal{C}$ are well behaved on all inputs, including the images of special points in $\mathcal{J}_\mathcal{C}$ under the `Project` map. The only time special points might be encountered is at the two end points of the scalar multiplication; but suitable point validation can detect and reject special points as inputs, and special outputs in $\mathcal{J}_\mathcal{C}$ are not the goal of an exceptional procedure attack—adversaries can only hope to trigger exceptional points early on in a scalar multiplication, where the number of possible intermediate points is subexponential.

6.2 Point recovery on the general Kummer

The general model of the Kummer surface $\mathcal{K}_\mathcal{C}^{\text{gen}}$ associated to $\mathcal{J}_\mathcal{C}$ is defined by

$$\mathcal{K}_\mathcal{C}^{\text{gen}} : K_2(\xi_1,\xi_2,\xi_3)\xi_4^2 + K_1(\xi_1,\xi_2,\xi_3)\xi_4 + K_0(\xi_1,\xi_2,\xi_3) = 0\ , \tag{4}$$

where

$$
\begin{aligned}
K_2 &= \xi_2^2 - 4\xi_1\xi_3, \\
K_1 &= -2(2f_0\xi_1^3 + f_1\xi_1^2\xi_2 + 2f_2\xi_1^2\xi_3 + f_3\xi_1\xi_2\xi_3 + 2f_4\xi_1\xi_3^2 + f_5\xi_2\xi_3^2 + 2f_6\xi_3^3), \\
K_0 &= (f_1^2 - 4f_0f_2)\xi_1^4 - 4f_0f_3\xi_1^3\xi_2 - 2f_1f_3\xi_1^3\xi_3 - 4f_0f_4\xi_1^2\xi_2^2 \\
&\quad + 4(f_0f_5 - f_1f_4)\xi_1^2\xi_2\xi_3 + (f_3^2 + 2f_1f_5 - 4f_2f_4 - 4f_0f_6)\xi_1^2\xi_3^2 - 4f_0f_5\xi_1\xi_2^3 \\
&\quad + 4(2f_0f_6 - f_1f_5)\xi_1\xi_2^2\xi_3 + 4(f_1f_6 - f_2f_5)\xi_1\xi_2\xi_3^2 - 2f_3f_5\xi_1\xi_3^3 - 4f_0f_6\xi_2^4 \\
&\quad - 4f_1f_6\xi_2^3\xi_3 - 4f_2f_6\xi_2^2\xi_3^2 - 4f_3f_6\xi_2\xi_3^3 + (f_5^2 - 4f_4f_6)\xi_3^4.
\end{aligned}
$$

6.3 Projection from Jacobians to general Kummers

`Project` implements the map $\mathcal{J}_\mathcal{C} \to \mathcal{K}_\mathcal{C}^{\text{gen}}$ described in [11, Eqs. (3.1.3–5)]; it maps generic points $\langle x^2 + a_1x + a_0, b_1x + b_0\rangle$ in $\mathcal{J}_\mathcal{C}$ to $(\xi_1 : \xi_2 : \xi_3 : \xi_4)$ in $\mathcal{K}_\mathcal{C}^{\text{gen}}$, where

$$\xi_1 = 1\ , \quad \xi_2 = -a_1\ , \quad \xi_3 = a_0\ , \quad \text{and} \tag{5}$$

$$\xi_4 = b_1^2 + (a_1^2 - a_0)\left(f_5a_1 - f_6(a_1^2 - a_0)\right) + a_1(f_3 - f_4a_1) - f_2\ . \tag{6}$$

This costs $5\mathbf{M} + 2\mathbf{S} + 7\mathbf{a}$ (saving $1\mathbf{M}$ if $f_6 = 0$).

6.4 Recovering Jacobian points from general Kummers

We now derive explicit formulæ for

$$\texttt{Recover} : (P, x(Q), x(Q \oplus P)) \longmapsto Q$$

in genus 2. We follow the approach of Okeya–Sakurai [31] and Brier–Joye [8] for elliptic curves, rewriting the equations used for computing $x(Q \oplus P)$ in terms of $P = (x(P), y(P))$ and $Q = (x(Q), y(Q))$, making $y(Q)$ the unknown. We may suppose that P and Q are nonzero and not points of order 2.

Suppose we are given[4]

$$
\begin{aligned}
P &= \langle x^2 + a_1(P)x + a_0(P), b_1(P)x + b_0(P)\rangle \in \mathcal{J}_\mathcal{C}(\mathbb{F}_q)\ , \\
x(Q) &= (1 : \xi_2 : \xi_3 : \xi_4) \in \mathcal{K}_\mathcal{C}^{\text{gen}}(\mathbb{F}_q)\ , \\
x(Q \oplus P) &= (1 : \xi_2^\oplus : \xi_3^\oplus : \xi_4^\oplus) \in \mathcal{K}_\mathcal{C}^{\text{gen}}(\mathbb{F}_q)\ ;
\end{aligned}
$$

we want to `Recover`

$$Q = \langle x^2 + a_1(Q)x + a_0(Q), b_1(Q)x + b_0(Q)\rangle \in \mathcal{J}_\mathcal{C}(\mathbb{F}_q)\ .$$

[4] We suppose that the inputs $x(Q)$, $x(Q \oplus P)$, and $x(Q \ominus P)$ are normalized here, to simplify the exposition. In practice, we use projectivized forms of these formulæ (which corresponds to replacing ξ_2, ξ_3, and ξ_4 with ξ_2/ξ_1, ξ_3/ξ_1, and ξ_4/ξ_1, etc.), in order to handle the non-normalized inputs that we encounter at the end of our addition chains. The input point P in $\mathcal{J}_\mathcal{C}$ can be presumed to be normalized, since it is the input to the scalar multiplication routine.

We already have $a_1(Q) = -\xi_2$ and $a_0(Q) = \xi_3$ from Eq. (5); it remains to compute $b_1(Q)$ and $b_0(Q)$.

Okeya and Sakurai noticed that the formulæ for y-coordinate recovery on Montgomery curves are simpler if $x(Q \ominus P)$ is also known (see [31, pp. 129–130]); we observed the same simplification in genus 2, where it was evident that computing $x(Q \ominus P)$ from $x(Q)$, $x(P)$ and $x(Q \oplus P)$ (using one more differential addition on the corresponding Kummer) yielded a faster overall recovery. We therefore begin our `Recover` with a call to `xADD`, to compute

$$x(Q \ominus P) = (1 : \xi_2^{\ominus} : \xi_3^{\ominus} : \xi_4^{\ominus}) = \texttt{xADD}(Q, P, x(Q \oplus P)) \ .$$

Since P and Q are nonzero, they correspond to unique degree-2 divisors on $\mathcal{C}$ (cf. [11, Ch. 1]):

$$P \longleftrightarrow [(u_P, v_P) + (u'_P, v'_P)] \ , \quad Q \longleftrightarrow [(u_Q, v_Q) + (u'_Q, v'_Q)] \ .$$

We do not compute the values of $u_P, v_P, u'_P, v'_P, u_Q, v_Q, u'_Q$, and v'_Q (which are generally defined over a quadratic extension): here they simply serve as formal devices, to aid our derivation of recovery formulæ. Let

$$\begin{aligned} G_1 &:= E_1 + E_2 \ , & G_2 &:= u'_P E_1 + u_P E_2 \ , \\ G_3 &:= E_3 + E_4 \ , & G_4 &:= u'_Q E_3 + u_Q E_4 \ , \end{aligned}$$

where

$$E_1 = \frac{v_P}{(u_P - u'_P)(u_P - u_Q)(u_P - u'_Q)} \ , \quad E_2 = \frac{v'_P}{(u'_P - u_P)(u'_P - u_Q)(u'_P - u'_Q)} \ ,$$
$$E_3 = \frac{v_Q}{(u_Q - u'_P)(u_Q - u_P)(u_Q - u'_Q)} \ , \quad E_4 = \frac{v'_Q}{(u'_Q - u_P)(u'_Q - u'_P)(u'_Q - u_Q)} \ .$$

The G_i are functions of P and Q, because they are symmetric with respect to $(u_P, v_P) \leftrightarrow (u'_P, v'_P)$ and $(u_Q, v_Q) \leftrightarrow (u'_Q, v'_Q)$; below, we will compute them as functions of P, $x(Q \oplus P)$, and $x(Q \ominus P)$. For notational convenience, we define

$$\begin{aligned} Z_1 &:= -(\xi_2 + a_1(P)) \ , & Z_2 &:= \xi_3 - a_0(P) \ , & (7) \\ Z_3 &:= a_1(P)\xi_3 + a_0(P)\xi_2 \ , & Z_4 &:= \xi_3 Z_2 - \xi_2 Z_3 \ , & (8) \end{aligned}$$

$$D := Z_2^2 - Z_1 Z_3 \ , \tag{9}$$

$$\Delta := -4G_2^2 + 2(\xi_2^{\oplus} + \xi_2^{\ominus})G_1 G_2 - 2(\xi_3^{\oplus} + \xi_3^{\ominus})G_1^2 \ , \tag{10}$$

and

$$\begin{aligned} T &:= f_6 - G_1^2 - G_3^2 \\ &= f_6 - G_1^2 - \frac{1}{D^2}\left(\begin{array}{c} \xi_4 D + f_0 Z_1^2 - f_1 Z_1 Z_2 + f_2 Z_2^2 - f_3 Z_2 Z_3 \\ + f_4 Z_3^2 - f_5 Z_3 Z_4 + f_6 Z_4^2 \end{array}\right) \ . \end{aligned} \tag{11}$$

We now use the fact that the cubic polynomial

$$\begin{aligned} l(x) &= E_1(x-u'_P)(x-u_Q)(x-u'_Q) + E_2(x-u_P)(x-u_Q)(x-u'_Q) \\ &\quad + E_3(x-u_P)(x-u'_P)(x-u'_Q) + E_4(x-u_P)(x-u'_P)(x-u_Q) \\ &= (G_1x - G_2)(x^2 + a_1(Q)x + a_0(Q)) + (G_3x - G_4)(x^2 - \xi_2 x + \xi_3) \end{aligned}$$

satisfies $\ell(x) \equiv b(x) \bmod a(x)$ when $\langle a(x), b(x)\rangle$ is the Mumford representation of P, Q or $\ominus(P \oplus Q)$ (this is just the geometric definition of the group law on $\mathcal{J}_\mathcal{C}$; the cubic $\ell(x)$ is analogous to the line through P, Q, and $\ominus(P \oplus Q)$ in the classic elliptic curve group law). Together with $b(x)^2 \equiv f(x) \pmod{a(x)}$, which is satisfied by the Mumford representation $\langle a(x), b(x)\rangle$ of every point on $\mathcal{J}_\mathcal{C}$, this gives the relations

$$\begin{pmatrix} b_1(Q) \\ b_0(Q) \end{pmatrix} = \begin{pmatrix} \xi_2 Z_1 + Z_2 & Z_1 \\ \xi_3 Z_1 & Z_2 \end{pmatrix} \begin{pmatrix} G_3 \\ G_4 \end{pmatrix} , \tag{12}$$

$$\begin{pmatrix} G_3 \\ G_4 \end{pmatrix} = \frac{T}{\Delta} \begin{pmatrix} \xi_3^{\ominus} - \xi_3^{\oplus} & \xi_2^{\oplus} - \xi_2^{\ominus} \\ \xi_2^{\oplus}\xi_3^{\ominus} - \xi_2^{\ominus}\xi_3^{\oplus} & \xi_3^{\oplus} - \xi_3^{\ominus} \end{pmatrix} \begin{pmatrix} G_1 \\ G_2 \end{pmatrix} , \tag{13}$$

and

$$\begin{pmatrix} G_1 \\ G_2 \end{pmatrix} = \frac{1}{D} \begin{pmatrix} Z_2 & -Z_1 \\ -a_0(P)Z_1 & a_1(P)Z_1 - Z_2 \end{pmatrix} \begin{pmatrix} b_1(P) \\ b_0(P) \end{pmatrix} . \tag{14}$$

Our `Recover` operation is now defined as follows: first, we use Eqs. (7) and (8) to compute the Z_i; then Eq. (9) yields D. Then we can use Eq. (14) to obtain G_1 and G_2, which we use to compute Δ and T using Eqs. (10) and (11). Equation (13) then yields G_3 and G_4, which we substitute into Eq. (12) to compute $b_1(Q)$ and $b_0(Q)$. We have thus recovered the full point

$$Q = \langle x^2 + a_1(Q)x + a_0(Q), b_1(Q)x + b_0(Q)\rangle$$

in $\mathcal{J}_\mathcal{C}$. Altogether, computing the map $(P, x(Q), x(Q \oplus P), x(Q \ominus P)) \mapsto Q$ costs $71\mathbf{M} + 8\mathbf{S} + 8\mathbf{m}_c + 35\mathbf{a} + 1\mathbf{I}$; adding the cost of computing $x(Q \ominus P)$ via an `xADD` operation, we obtain the full cost of computing $\texttt{Recover}(P, x(Q), x(Q \oplus P))$.

Remark 5. It is worth noting that $\xi_4^{\oplus}$ and $\xi_4^{\ominus}$ do not appear in our recovery procedure; this may be useful in scenarios where it is advantageous to omit their computation or transmission.

6.5 Scalar multiplication on Jacobians via general Kummers

The use of general Kummers in cryptography was investigated by Smart and Siksek [34] and Duquesne [15]. While they represent a natural generalization of x-only arithmetic for elliptic curves, in their full generality they do not offer competitive performance. The `xADD` and `xDBL` operations are defined by complicated biquadratic forms in the ξ_i (see [11]), which are hard to evaluate quickly for general curve parameters. While these formulæ are completely compatible with our

`Project`-pseudomultiply-`Recover` pattern, and yield scalar multiplication algorithms that may be useful to number theorists, the cryptographic applications of these algorithms are *à priori* limited. We will therefore skip any investigation of these scalar multiplication algorithms here, and move on directly to Gaudry's fast Kummer surfaces.

We remark, nevertheless, that the use of these biquadratic forms in conjunction with the `Project` and `Recover` defined above yields a solution to the problem of defining uniform and constant-time scalar multiplication algorithms for general genus 2 Jacobians. We leave the eventual optimization of these algorithms as an open problem, along with their application to Jacobians with special endomorphism structure, or Jacobians whose Kummers can be put into intermediate forms between the general and fast models.

7 Efficient Scalar Multiplication via Fast Kummers

While the performance of general Kummer models is somewhat disappointing, Gaudry [19] showed if we allow a certain 2-torsion structure on the Jacobian, then an alternative classical model for the Kummer (investigated algorithmically by Chudnovsky and Chudnovsky [12]) yields a dramatic speedup, competitive with—and regularly outperforming—elliptic curve arithmetic. Jacobians equipped with these "fast" Kummers are ideal candidates for our techniques; here we use the model with squared theta coordinates described in [13, Ch. 4].

7.1 Construction of fast Kummers

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

$$\begin{aligned} A &:= a+b+c+d\ , & B &:= a+b-c-d\ , \\ C &:= a-b+c-d\ , & D &:= a-b-c+d \end{aligned}$$

are nonzero, and

$$e = \frac{1+\alpha}{1-\alpha} f \qquad \text{where} \qquad \alpha^2 = \frac{CD}{AB}.$$

We set

$$\lambda = \frac{ac}{bd}\ , \qquad \mu = \frac{ce}{df}\ , \qquad \nu = \frac{ae}{bf}\ ,$$

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

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

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

$$\mathcal{K}_{\mathcal{C}}^{\text{fast}} : \begin{pmatrix} (X^2+Y^2+Z^2+T^2) \\ -F(XT+YZ) - G(XZ+YT) - H(XY+ZT) \end{pmatrix}^2 = EXYZT \quad (15)$$

where

$$\begin{aligned} E &:= 4abcd\,(ABCD/((ad-bc)(ac-bd)(ab-cd)))^2 \ , \\ F &:= (a^2-b^2-c^2+d^2)/(ad-bc) \ , \\ G &:= (a^2-b^2+c^2-d^2)/(ac-bd) \ , \\ H &:= (a^2+b^2-c^2-d^2)/(ab-cd) \ . \end{aligned}$$

For efficient arithmetic on $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$, we precompute the *theta constants*

$$\begin{aligned} x_0 &:= 1 \ , & y_0 &:= a/b \ , & z_0 &:= a/c \ , & t_0 &:= a/d \ , \\ x'_0 &:= 1 \ , & y'_0 &:= A/B \ , & z'_0 &:= A/C \ , & t'_0 &:= A/D \ . \end{aligned}$$

The image of the identity element of $\mathcal{J}_{\mathcal{C}}$ in $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$ is

$$(a\colon b\colon c\colon d) = (1/x_0\colon 1/y_0\colon 1/z_0\colon 1/d_0) \ ;$$

we also observe that $(A : B : C : D) = (1/x'_0 : 1/y'_0 : 1/z'_0 : 1/d'_0)$.

7.2 Projection from Jacobians to fast Kummers

`Project` implements the map $\mathcal{J}_{\mathcal{C}} \to \mathcal{K}_{\mathcal{C}}^{\text{fast}}$ from [13, §5.3], which is defined for generic points in $\mathcal{J}_{\mathcal{C}}$ by $\langle x^2 + a_1 x + a_0, b_1 x + b_0\rangle \mapsto (X\colon Y\colon Z\colon T)$, where

$$\begin{aligned} X &= a\left(a_0(\mu - a_0)(\lambda + a_1 + \nu) - b_0^2\right) \ , \\ Y &= b\left(a_0(\nu\lambda - a_0)(1 + a_1 + \mu) - b_0^2\right) \ , \\ Z &= c\left(a_0(\nu - a_0)(\lambda + a_1 + \mu) - b_0^2\right) \ , \\ T &= d\left(a_0(\mu\lambda - a_0)(1 + a_1 + \nu) - b_0^2\right) \ . \end{aligned}$$

This costs $11\mathbf{M} + 1\mathbf{S} + 3\mathbf{m}_c + 12\mathbf{a} + 1\mathbf{I}$, assuming the output point is normalized with $X = 1$.

For special points $\langle x - \alpha, \beta\rangle$ in $\mathcal{J}_{\mathcal{C}}$, `Project` is defined by first adding a point of order 2 in $\mathcal{J}_{\mathcal{C}}$, e.g., adding $\langle x - \lambda, 0\rangle$ to get $\langle x^2 - (\alpha+\beta)x + \alpha\beta, \frac{\beta}{\alpha-\lambda}x - \frac{\beta\lambda}{\alpha-\lambda}\rangle$, then using the above map to $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$, where the translation is undone by using the coordinate permutation-and-negation that corresponds to translation by $\langle x - \lambda, 0\rangle$ (see [19, §3.4]). Finally, $0_{\mathcal{J}_{\mathcal{C}}}$ (whose Mumford representation is $\langle 1, 0\rangle$) maps to $(a : b : c : d)$.

7.3 Basic fast Kummer arithmetic

We recall the formulæ and operation counts for `xDBL` and `xADD` on fast Kummers from [19, §3.2]. If

$$\begin{aligned} x(P) &= (X_1 : Y_1 : Z_1 : T_1) \ , \\ x(Q) &= (X_2 : Y_2 : Z_2 : T_2) \ , \\ x(P \ominus Q) &= (X_\ominus : Y_\ominus : Z_\ominus : T_\ominus) \ , \end{aligned}$$

then `xADD` maps $(x(P), x(Q), x(P \ominus Q))$ to $(X_\oplus : Y_\oplus : Z_\oplus : T_\oplus)$, where

$$X_\oplus = (X' + Y' + Z' + Y')^2/X_\ominus \ , \qquad Y_\oplus = (X' + Y' - Z' - Y')^2/Y_\ominus \ ,$$
$$Z_\oplus = (X' - Y' + Z' - Y')^2/Z_\ominus \ , \qquad T_\oplus = (X' - Y' - Z' + Y')^2/T_\ominus \ ,$$

where

$$\begin{aligned} X' &= x'_0(X_1 + Y_1 + Z_1 + T_1)(X_2 + Y_2 + Z_2 + T_2) \ , \\ Y' &= y'_0(X_1 + Y_1 - Z_1 - T_1)(X_2 + Y_2 - Z_2 - T_2) \ , \\ Z' &= z'_0(X_1 - Y_1 + Z_1 - T_1)(X_2 - Y_2 + Z_2 - T_2) \ , \\ T' &= t'_0(X_1 - Y_1 - Z_1 + T_1)(X_2 - Y_2 - Z_2 + T_2) \ . \end{aligned}$$

We can compute `xADD` using 8 squares, 7 products (3 by constants, ignoring the multiplication by $x'_0 = 1$), and 4 divisions (or, projectively, another 10 products); but in our applications, $(X_\ominus : Y_\ominus : Z_\ominus : T_\ominus)$ is fixed. so we can precompute $1/X_\ominus$, $1/Y_\ominus$, $1/Z_\ominus$, and $1/T_\ominus$, and then the 4 divisions become 4 products.

The `xDBL` operation is defined by exactly the same formulæ on setting $(X_2 : Y_2 : Z_2 : T_2) = (X_1 : Y_1 : Z_1 : T_1)$ and $(X_\ominus : Y_\ominus : Z_\ominus : T_\ominus) = (a : b : c : d)$ (so $X' = x'_0(X_1 + Y_1 + Z_1 + T_1)^2$, and so on).

The combined `xDBLADD` operation is outlined in Gaudry [19, §3.3]: we share the computation of $x'_0(X_1+Y_1+Z_1+T_1)^2$, etc., between the calculation of `xDBL` and `xADD`. The resulting operation costs 16 products and 1 square.

Finally, the function `ADD`: $P, Q \to x(Q \oplus P)$ in $\mathcal{J}_\mathcal{C}$ is computed via the formulæ in [23, Eq. (12)] at a cost of $22\mathbf{M} + 2\mathbf{S} + 1\mathbf{I} + 27\mathbf{a}$.

7.4 Recovering Jacobian points from fast Kummers

The `Recover` operation for fast Kummers uses the `Recover` for general Kummers (defined in §6.2) as a subroutine. To move between the fast Kummer $\mathcal{K}_\mathcal{C}^{\text{fast}}$ and the general Kummer $\mathcal{K}_\mathcal{C}^{\text{gen}}$, we modify the map from $\mathcal{J}_\mathcal{C}$ to $\mathcal{K}_\mathcal{C}^{\text{fast}}$ given in [13, §5.3] (which was in turn derived from [36]). This gives a linear isomorphism

$$\tau : \mathcal{K}_\mathcal{C}^{\text{gen}} \longrightarrow \mathcal{K}_\mathcal{C}^{\text{fast}} \ ,$$

defined on generic points by

$$\tau : (\xi_1 : \xi_2 : \xi_3 : \xi_4) \longmapsto (X : Y : Z : T) = (\xi_1 : \xi_2 : \xi_3 : \xi_4) M(a, b, c, d)^t \ ,$$

where

$$M(a, b, c, d) = \begin{pmatrix} \mu(\lambda + \nu) & \nu\lambda(1 + \mu) & \nu(\lambda + \mu) & \mu\lambda(1 + \nu) \\ -\mu & -\nu\lambda & -\nu & -\mu\lambda \\ \mu + 1 & \lambda + \nu & \nu + 1 & \lambda + \mu \\ -1 & -1 & -1 & -1 \end{pmatrix} \ .$$

The inverse map $\tau^{-1} : \mathcal{K}_\mathcal{C}^{\text{fast}} \to \mathcal{K}_\mathcal{C}^{\text{gen}}$ is defined by the matrix $(M(a, b, c, d)^t)^{-1}$. Since evaluating these linear transformations only involves multiplications by

constants, evaluating τ to map a point from $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$ to $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$ (or τ^{-1} to map a point from $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$ back to $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$) costs $16\mathbf{M} + 12\mathbf{a}$. Since the coordinates on both $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$ and $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$ are projective, we can save $1\mathbf{M}$ by scaling one of the entries in the transformation matrix to 1.

With $x\colon \mathcal{J}_{\mathcal{C}} \to K$, the operation `Recover`: $P, x(Q), x(Q \oplus P) \mapsto Q$ is computed as follows. As was done in §6.2, the first step is to compute $x(Q \ominus P)$ from $x(P)$, $x(Q)$ and $x(Q \oplus P)$ via an `xADD` operation (note that $x(P)$ was already obtained during the `Project` operation). We then use τ^{-1} to map all four elements, $x(P)$, $x(Q)$, $x(Q \oplus P)$ and $x(Q \ominus P)$ to the corresponding general Kummer $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$. We can then use the `Recover` operation on $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$ to output a normalized point Q in Mumford coordinates using $71\mathbf{M} + 4\mathbf{m}_c + 8\mathbf{S} + 34\mathbf{a} + 1\mathbf{I}$. Recall from §6.2 that the fourth Kummer coordinate ξ_4 is only needed for the Kummer point corresponding to Q. Thus, the map τ^{-1} is only computed in full once (costing $15\mathbf{M} + 12\mathbf{a}$), but for the three other points can omit the fourth coordinate (costing $11\mathbf{M} + 9\mathbf{a}$).

In total, including the initial `xADD` operation (costing $14\mathbf{M}+4\mathbf{S}+3\mathbf{m}_c+12\mathbf{a}$), the map `Recover`: $P, x(Q), x(Q \oplus P) \mapsto Q$ when $x\colon \mathcal{J}_{\mathcal{C}} \to \mathcal{K}_{\mathcal{C}}^{\text{fast}}$ costs $133\mathbf{M} + 12\mathbf{S} + 7\mathbf{m}_c + 97\mathbf{a} + 1\mathbf{I}$. Here we count multiplications by the curve constants f_i as full multiplications.

7.5 Scalar multiplication on Jacobians with fast Kummers

The costs of our key subroutines for fast Kummers are summarized in Table 5.

Table 5. Costs of operations for fast Kummers, where $\mathbf{m}_c$ is used to denote multiplications by the theta constants (x_0, x_0', etc) or by the curve constants (f_i). Normalization refers to the respective projective coordinates.

Algorithm	**M**	**S**	$\mathbf{m}_c$	**a**	**I**	Conditions
`ADD`	22	2	0	27	1	P and Q normalized
`Project`	11	1	3	12	1	—
`xDBL`	0	8	6	16	0	—
`xADD`	14	4	3	24	0	—
`xADD`	7	4	3	24	0	$x(Q \ominus P)$ fixed
`xDBLADD`	17	9	6	32	0	—
`xDBLADD`	10	9	6	32	0	$x(Q \ominus P)$ fixed
`Recover`	133	12	7	97	1	P normalized

Theorem 4. *Let $\mathcal{J}_{\mathcal{C}}$ be the Jacobian of a genus 2 curve admitting a fast Kummer surface, as in §7.1.*

1. *Let P be a point in $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q) \setminus \mathcal{J}_{\mathcal{C}}[2]$, and let m be a positive β-bit integer. Then Algorithm 1 computes $[m]P$ in*

$$(10\beta + 134)\mathbf{M} + (9\beta + 12)\mathbf{S} + (6\beta + 10)\mathbf{m}_c + (32\beta + 93)\mathbf{a} + 2\mathbf{I} \ .$$

2. *Let P and Q be points in $\mathcal{J}_{\mathcal{C}}(\mathbb{F}_q) \setminus \mathcal{J}_{\mathcal{C}}[2]$, and let m and n be positive β-bit integers. Then Algorithm 3 computes $[m]P \oplus [n]Q$ in*

$$(17\beta + 194)\mathbf{M} + (13\beta + 17)\mathbf{S} + (9\beta + 16)\mathbf{m}_c + (56\beta + 160)\mathbf{a} + 2\mathbf{I} \ .$$

Proof. Take the values from Table 5 in Lemma 1 for the first part and Theorem 1 for the second, using a simultaneous inversion [29] to replace the $3\mathbf{I}$ (corresponding to the 3 `Project` operations) by $6\mathbf{M} + 1\mathbf{I}$. □

Remark 6. As we noted in Remark 1, similar techniques appear in Lubicz and Robert [28] for general abelian varieties in higher dimension embedded in projective space via theta functions. After suitable precomputations, their "compatible addition" operation views the results of Montgomery-like pseudomultiplication algorithms on a Kummer variety K of dimension g as points on the corresponding abelian variety A embedded in $K \times K$, and therefore chooses the "correct" result of an addition on K. They explain how to map the resulting point into the 4Θ embedding of A (in $\mathbb{P}^{4^g-1}$; for genus 2, this is $\mathbb{P}^{15}$). A major difference with our treatment here is that Robert and Lubicz cannot treat A as a Jacobian (since general abelian varieties of dimension $g > 3$ are not Jacobians); hence, when specializing to genus 2, there is no connection with any curve $\mathcal{C}$ or Jacobian $\mathcal{J}_{\mathcal{C}}$ (and in particular, the starting and finishing points do not involve the Mumford representation). Kohel [25] explores similar ideas for elliptic curves, leading to a very interesting interpretation of Edwards curve arithmetic.

8 Applications to signatures

Smart and Siksek [34] first showed that the action of $\mathbb{Z}$ on $\mathcal{G}/\langle\pm 1\rangle$ could be used to instantiate Diffie–Hellman on $\mathcal{G}/\langle\pm 1\rangle$ rather than $\mathcal{G}$. In [34, §5], they pondered whether protocols like ElGamal encryption could be instantiated on $\mathcal{G}/\langle\pm 1\rangle$, and observed that the main obstruction appeared to be that such protocols require an addition in the group—that is, a true group structure and not a mere $\mathbb{Z}$-action.

Our `Project`-pseudomultiply-`Recover` technique allows a range of cryptographic protocols beyond Diffie–Hellman key exchange to take advantage of the fast and uniform (multi-)scalar multiplications offered on $\mathcal{G}/\langle\pm 1\rangle$, where $\mathcal{G}$ can now be any elliptic curve *or* any genus 2 Jacobian (defined over a large characteristic field).

In this section we illustrate this by showing how Schnorr signatures [33] can take advantage of the fast arithmetic available when $\mathcal{G}/\langle\pm 1\rangle$ is the fast Kummer surface $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$. More specifically, we describe an instantiation of a Schnorr signature scheme where $\mathcal{G}$ is the Jacobian of the Gaudry–Schost curve [21] used recently to set Diffie–Hellman speed records [4] at the 128-bit security level. Following the design of EdDSA signatures [5], we hash both the message M and signer's public key Q alongside the first half of the signature. We emphasise, however, that both of these choices are merely for illustrative purposes, and that any known variant of ElGamal signatures (Schnorr or otherwise) can use $\mathcal{G}/\langle\pm 1\rangle$ for any suitable choice of $\mathcal{G}$.

8.1 The Gaudry-Schost Jacobian

Let $q = 2^{127} - 1$, choose an α in $\mathbb{F}_q$ such that $363\alpha^2 + 833 = 0$, define the six constants

$$a = 11\ , \quad b = -22\ , \quad c = -19\ , \quad c = -3\ , \quad e = 1 + \alpha\ , \quad f = 1 - \alpha\ ;$$

setting $\lambda := (ac)/(bd)$, $\mu := (ce)/(df)$, and $\nu := (ae)/(bf)$, we let $\mathcal{C}$ be the genus 2 curve over $\mathbb{F}_q$ defined by

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

The Jacobian $\mathcal{J}_\mathcal{C}$ of $\mathcal{C}$ has cardinality $\#\mathcal{J}_\mathcal{C}(\mathbb{F}_q) = 2^4 N$, where

$$N = 2^{250} - \texttt{0x334D69820C75294D2C27FC9F9A154FF47730B4B840C05BD}$$

is a 250-bit prime.

We define the 127-bit encoding of $\mathbb{F}_q$ to be the little-endian encoding of $\{0, 1, \ldots, 2^{127} - 2\}$, and let $\mathcal{H}\colon \{0,1\}^* \to \{0,1\}^{512}$ be any suitable hash function with a 512-bit output (eg. SHA-512 or SHA3-512).

To ensure that all our scalars are positive and of fixed bitlength, when computing scalar multiplications $[m]P$ on $\mathcal{J}_\mathcal{C}$ we systematically replace m with $m' := (m \bmod N) + 3N$; then m' always has exactly 252 bits (since both $3N$ and $4N$ have 252 bits). Following the lead of [5], the cofactor 16 is included in the key generation and verification operations to void any potential threat of small subgroup attacks [26]. In this case, the scalar $16z$ is parsed by appending four zeroes to the parsing of z described above, so a multiplication by the 252-bit scalar z is followed by 4 doubling operations. Where applicable, the operation counts stated below include the cost of these 4 additional `DBL` or `xDBL` operations.

To compare our `Project`-pseudomultiply-`Recover` approach to traditional arithmetic in $\mathcal{J}_\mathcal{C}$, we take the operation counts from [23, Table 2], where `ADD` costs $41\mathbf{M}+7\mathbf{S}+22\mathbf{a}$, `DBL` costs $26\mathbf{M}+8\mathbf{S}+2\mathbf{m}_c+25\mathbf{a}$, `mADD` costs $32\mathbf{M}+5\mathbf{S}+22\mathbf{a}$ and `mDBLADD` costs $57\mathbf{M}+8\mathbf{S}+42\mathbf{a}$. Here `mADD` and `mDBLADD` compute $P \oplus Q$ and $[2]P \oplus Q$ where P is projective and Q (which is typically a fixed lookup table element) is normalized. We assume a fixed, signed window approach to computing one-dimensional scalar multiplications in $\mathcal{J}_\mathcal{C}$; our experiments with 252-bit scalars found the window width $w = 5$ to be optimal. In this case the scalar multiplications need $8 \times$ `DBL` $+ 7 \times$ `mADD` operations to build the lookup table of 16 elements, $1\mathbf{I}+39\mathbf{M}$ to normalize its entires, $200 \times$ `DBL` $+ 50 \times$ `mDBLADD` operations in the main loop, and $1\mathbf{I}+4\mathbf{M}$ to normalize the output. For the two-dimensional scalar multiplications, we assume the standard approach to such multiexponentiations (cf. [18, §3]). Our experiments showed that $w = 2$ is optimal here, meaning the lookup table contains 16 elements $[u]P + [v]Q$ for $0 \le u, v \le 3$. In this case two-dimensional scalar multiplications need $3 \times$ `DBL` $+ 10 \times$ `mADD` operations to build the lookup table, $1\mathbf{I}+36\mathbf{M}$ to normalize its entries, $125 \times$ `DBL` $+ 125 \times$ `mDBLADD` operations in the main loop, and $1\mathbf{I} + 4\mathbf{M}$ to normalize the output.

We only present counts for the dominant operations, i.e., the (multi)scalar multiplications, below. For the exact costs of compression and decompression, see

Appendix A; we omit details of point validation in $\mathcal{J}_\mathcal{C}$ and $\mathcal{K}_\mathcal{C}^{\text{fast}}$, both of which are essentially for free. For simplicity, we compare key generation, signing, and verification operations in §8.2 assuming that no precomputation is performed offline, noting that, if space permits it, all of these operations can take advantage of precomputation in practice.

8.2 Schnorr signatures on the Gaudry-Schost Jacobian

Let the public generator, P, be any point of order N in $\mathcal{J}_\mathcal{C}(\mathbb{F}_q)$. Including P, generic elements $\langle x^2 + a_1 x + a_0, b_1 x + b_0 \rangle$ in $\mathcal{J}_\mathcal{C}(\mathbb{F}_q)$ are compressed as in §A and encoded as a 256-bit string $(\texttt{bit}_0 || a_0 || \texttt{bit}_1 || a_1)$.

Key generation: Given the public generator P and a 256-bit secret key d, compute $\mathcal{H}(d) = (d' || d'')$, where d' and d'' are both 256-bit strings. The public key is computed as the 256-bit encoding of $Q = [16d']P$. Computing $(d', P) \mapsto Q$ directly on the Jacobian costs $8629\mathbf{M} + 2131\mathbf{S} + 424\mathbf{m}_c + 7554\mathbf{a} + 2\mathbf{I}$, but using Algorithm 1 to project, pseudomultiply on $\mathcal{K}_\mathcal{C}^{\text{fast}}$, and recover, costs only $2654\mathbf{M} + 2312\mathbf{S} + 1546\mathbf{m}_c + 8221\mathbf{a} + 2\mathbf{I}$ – see Theorem 4.

Signing: Given the public generator P, a message M, and the secret key $\mathcal{H}(d) = (d' || d'')$, compute $r = \mathcal{H}(d'' || M)$, then the 256-bit encoding of $R = [r]P$, then $h = \mathcal{H}(R || Q || M)$, and finally the 256-bit encoding of $s = (r - 16hd') \bmod N$. The signature is the 512-bit string $(R || s)$. The costs of performing $(r, P) \mapsto [r]P$ are almost the same as above, except for the cost of the four final doublings in both scenarios: on $\mathcal{J}_\mathcal{C}$ this comes to $8525\mathbf{M} + 2099\mathbf{S} + 416\mathbf{m}_c + 7454\mathbf{a} + 2\mathbf{I}$, while using Algorithm 1 costs $2654\mathbf{M} + 2280\mathbf{S} + 1522\mathbf{m}_c + 8157\mathbf{a} + 2\mathbf{I}$.

Verification: Given the public generator P, a message M, and a putative signature $(R || s)$ on M associated with a public key Q, compute $h = \mathcal{H}(R || Q || M)$ and accept the signature if $[16s]P + [16h]Q = [16]R$; otherwise, reject. Computing $((s, h), (P, Q)) \mapsto [16]([s]P + [h]Q)$ directly on the Jacobian with the costs $10813\mathbf{M} + 2074\mathbf{S} + 256\mathbf{m}_c + 8670\mathbf{a} + 2\mathbf{I}$. Using Algorithm 3 to perform the 2-dimensional scalar multiplication via $\mathcal{K}_\mathcal{C}^{\text{fast}}$ costs $4478\mathbf{M} + 3325\mathbf{S} + 2308\mathbf{m}_c + 14272\mathbf{a} + 2\mathbf{I}$ – see Theorem 4.

We summarize the costs of key generation, signing and verification in Table 6. In addition to the speed benefits of Algorithm 1 and Algorithm 3 in this scenario, we reiterate that their working on $\mathcal{K}_\mathcal{C}^{\text{fast}}$ avoids the side-channel issues discussed in §6.1.

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*

Table 6. Costs of key generation, signing and verification in the above Schnorr signature scheme.

Algorithm	Method	**M**	**S**	$\mathbf{m}_c$	**a**	**I**
Key Generation	fixed window mult. on $\mathcal{J}_\mathcal{C}$	8629	2131	424	7554	2
	Algorithm 1	2654	2312	1546	8221	2
Signing	fixed window mult. on $\mathcal{J}_\mathcal{C}$	8525	2099	416	7454	2
	Algorithm 1	2654	2280	1522	8157	2
Verification	multiscalar mult. on $\mathcal{J}_\mathcal{C}$	10813	2074	256	8670	2
	Algorithm 3	4478	3325	2308	14272	2

- *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. Daniel J. Bernstein. Differential addition chains. preprint, 2006.
3. Daniel J. Bernstein. Elliptic vs. hyperelliptic, part I. Talk at ECC 2006, Fields Institute, Toronto, Canada, 2006. http://cr.yp.to/talks/2006.09.20/slides-djb-20060920-a4.pdf.
4. Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Peter Schwabe. Kummer strikes back: New DH speed records. In Sarkar and Iwata [32], pages 317–337.
5. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89, 2012.
6. Daniel J. Bernstein and Tanja Lange. Explicit formulas database, 2015. http://www.hyperelliptic.org/EFD/.
7. Brainpool. ECC Brainpool standard curves and curve generation v. 1.0, 2005. http://www.ecc-brainpool.org/download/Domain-parameters.pdf.
8. 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.
9. Daniel R. L. Brown. Multi-dimensional montgomery ladders for elliptic curves. 2006.http://eprint.iacr.org/2006/220.
10. David G. Cantor. Computing in the Jacobian of a hyperelliptic curve. *Mathematics of computation*, 48(177):95–101, 1987.
11. J. W. S. Cassels and E. V. Flynn. *Prolegomena to a middlebrow arithmetic of curves of genus 2*, volume 230. Cambridge University Press, 1996.
12. 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.
13. Romain Cosset. *Applications of theta functions for hyperelliptic curve cryptography.* Ph.D Thesis, Université Henri Poincaré - Nancy I, November 2011.
14. 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.
15. 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.
16. Harold Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44(3):393–422, 2007.
17. Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.
18. 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.
19. Pierrick Gaudry. Fast genus 2 arithmetic based on Theta functions. *J. Mathematical Cryptology*, 1(3):243–265, 2007.
20. Pierrick Gaudry and David Lubicz. The arithmetic of characteristic 2 Kummer surfaces and of elliptic Kummer lines. *Finite Fields and Their Applications*, 15(2):246–260, 2009.
21. Pierrick Gaudry and Eric Schost. Genus 2 point counting over prime fields. *J. Symb. Comput.*, 47(4):368–400, 2012.
22. Mike Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. http://eprint.iacr.org/.
23. Huseyin Hisil and Craig Costello. Jacobian coordinates on genus 2 curves. In Sarkar and Iwata [32], pages 338–357.
24. 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.
25. 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.
26. Chae Hoon Lim and Pil Joong Lee. A key recovery attack on discrete log-based schemes using a prime order subgroupp. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 249–263. Springer, 1997.
27. 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.
28. David Lubicz and Damien Robert. Arithmetic on abelian and kummer varieties. *IACR Cryptology ePrint Archive*, 2014:493, 2014.

29. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.
30. NIST. Recommended elliptic curves for federal government use, 1999. http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf.
31. 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.
32. 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.
33. Claus-Peter Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174, 1991.
34. Nigel P. Smart and Samir Siksek. A fast Diffie–Hellman protocol in genus 2. *Journal of cryptology*, 12(1):67–73, 1999.
35. Colin Stahlke. Point compression on Jacobians of hyperelliptic curves over $\mathbb{F}_q$. *IACR Cryptology ePrint Archive*, 2004:30, 2004.
36. Paul van Wamelen. Equations for the Jacobian of a hyperelliptic curve. *Transactions of the American Mathematical Society*, 350(8):3083–3106, 1998.

A Point compression in genus 2

Here we show how to compress points on $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$, $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$ and $\mathcal{J}_{\mathcal{C}}$. We use $\mathbf{E}$ to denote a fixed exponentiation in the underlying finite field, e.g., to compute a square root. It is commonly the case that $\mathbf{E} \approx \mathbf{I}$.

Compressing points on the general Kummer $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$. Generic points $P = (\xi_1 : \xi_2 : \xi_3 : \xi_4)$ in $\mathcal{K}_{\mathcal{C}}^{\text{gen}}(\mathbb{F}_q) \subset \mathbb{P}^3(\mathbb{F}_q)$ have $\xi_1 \neq 0$, so we can compute a normalized representative $(1 : k_2 : k_3 : k_4)$, where each $k_i = \xi_i/\xi_1$, at a cost of $3\mathbf{M} + 1\mathbf{I}$. We can further compress P to the data $(k_2, k_3, \texttt{bit})$, where `bit` is a single bit, as follows: the defining equation of $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$ (Eq. (4)) is quadratic in ξ_4, so

$$k_4 = \frac{-K_1(1,k_2,k_3) \pm \sqrt{K_1(1,k_2,k_3)^2 - 4K_0(1,k_2,k_3)K_2(1,k_2,k_3)}}{2K_2(1,k_2,k_3)}\ , \qquad (16)$$

which means that k_4 can be recovered (during decompression) from k_2, k_3, and the "sign" of the square root. To compress, after computing (k_2, k_3, k_4), we compute $K_1 = K_1(1, k_2, k_3)$ and $K_2 = K_2(1, k_2, k_3)$ and set `bit` to be the sign bit[5] of $2K_2k_4 + K_1$. Computing the sign bit costs $7\mathbf{M} + 2\mathbf{S} + 11\mathbf{a}$.

To decompress $(k_2, k_3, \texttt{bit})$ to a point on $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$, we first compute $K_0 = K_0(1, k_2, k_3)$, $K_1 = K_1(1, k_2, k_3)$ and $K_2 = K_2(1, k_2, k_3)$. We can then use a

[5] When q is a large prime, the sign bit of an element is typically chosen as its parity when represented as an integer in $[0, q-1)$.

simultaneous inversion-and-square-root[6] to compute both $\sqrt{K_1^2 - 4K_0K_2}$ and $1/(2K_2)$ using one field exponentiation, before recovering k_4 via (16). The decompressed point is then $(1 : k_2 : k_3 : k_4)$; the decompression costs $25\mathbf{M} + 4\mathbf{S} + 24\mathbf{a} + 1\mathbf{E}$.

Compressing points on the fast Kummer $\mathcal{K}_{\mathcal{C}}^{\mathbf{fast}}$. As with the general Kummer $\mathcal{K}_{\mathcal{C}}^{\text{gen}}$, generic points $P = (X : Y : Z : T)$ in $\mathcal{K}_{\mathcal{C}}^{\text{fast}}(\mathbb{F}_q) \subset \mathbb{P}^3(\mathbb{F}_q)$ have a nonzero first coordinate X, so we might begin compressing P by normalizing the X-coordinate to 1. But the defining equation of $\mathcal{K}_{\mathcal{C}}^{\text{fast}}$ (Eq. (15)) is quartic in all of its variables, which suggests that compressing another coordinate would require solving an (unwieldy) during decompression. A faster approach to achieving further compression[7] is to map $(X : Y : Z : T)$ to the general Kummer, and then to normalize before performing compression/decompression there. The cost of compressing $P = (X : Y : Z : T)$ into two $\mathbb{F}_q$ elements and a single bit is therefore $25\mathbf{M} + 2\mathbf{S} + 23\mathbf{a} + 1\mathbf{I}$, and the cost of decompression $40\mathbf{M} + 4\mathbf{S} + 36\mathbf{a} + 1\mathbf{E}$.

Compressing points on the Jacobian. To compress the four Mumford coordinates, we follow Stahlke's technique [35]. For general genus 2 curves, compression of $\langle x^2 + a_1x + a_0, b_1x + b_0\rangle$ into $(a_1, a_0, \mathtt{bit}_1, \mathtt{bit}_0)$ costs $3\mathbf{M} + 1\mathbf{S} + 4\mathbf{a}$, while decompression costs $36\mathbf{M} + 5\mathbf{S} + 45\mathbf{a} + 2\mathbf{E}$. Compression costs the same for genus 2 curves in Rosenhain form, but decompression is significantly faster and requires $18\mathbf{M} + 4\mathbf{S} + 27\mathbf{a} + 2\mathbf{E}$. In the latter case, we compress by setting $\mathtt{bit}_1$ and $\mathtt{bit}_0$ as the least significant bits of $4(a_1b_1b_0 - a_0b_1^2 - b_0^2)$ and b_1 respectively. For decompression, we recover b_1 and b_0 from $(a_1, a_0, \mathtt{bit}_1, \mathtt{bit}_0)$ by first computing

$$A = a_1^2 - 4a_0, \qquad C = (a_0(a_0 - f_3 - a_1(a_1 - f_4)) + f_1)^2, \quad \text{and}$$
$$B = 2(f_4a_0(2a_0 - a_1^2) + a_0(f_3a_1 - 2f_2) + a_1(f_1 + a_0(a_1^2 - 3a_0))),$$

and using $\mathtt{bit}_1$ to choose z_0 as the correct root $z_0 = \sqrt{B^2 - 4AC}$. We then set $z_1 = (z_0 - B)/(2A)$; here the square root and the inversion of $2A$ can again be combined into one exponentiation. We can then recover b_1 as $b_1 = \sqrt{f_4(a_1^2 - a_0) - a_1(f_3 + a_1^2 - 2a_0) + f_2 + z_1}$, using $\mathtt{bit}_0$ to choose the root. Finally, we recover b_0 as $b_0 = (a_0(f_4a_1 - f_3 - q^2 + a_0) + qz_1 + f_1)/(2b_1)$, noting again that the square root and the inversion can be computed simultaneously.

[6] E.g., compute $\sqrt{u}$ and $1/v$ via $w \leftarrow uv^2$, $w \leftarrow w^{-1/2}$, then $(\sqrt{u}, 1/v) \leftarrow (uvw, uw)$.
[7] This faster compression/decompression answers a question posed by Bernstein [3].