Kummer for Genus One over Prime Order Fields*

Sabyasachi Karati[1] and Palash Sarkar[2]

[1] iCIS Lab
Department of Computer Science
University of Calgary
email: sabyasachi.karati@ucalgary.ca**
[2] Applied Statistics Unit
Indian Statistical Institute
203, B.T. Road, Kolkata
India 700108.
e-mail: palash@isical.ac.in

Abstract. This work considers the problem of fast and secure scalar multiplication using curves of genus one defined over a field of prime order. Previous work by Gaudry and Lubicz in 2009 had suggested the use of the associated Kummer line to speed up scalar multiplication. In the present work, we explore this idea in detail. The first task is to obtain an elliptic curve in Legendre form which satisfies necessary security conditions such that the associated Kummer line has small parameters and a base point with small coordinates. It turns out that the ladder step on the Kummer line supports parallelism and can be implemented very efficiently in constant time using the single-instruction multiple-data (SIMD) operations available in modern processors. For the 128-bit security level, this work presents three Kummer lines denoted as $K_1 := \mathsf{KL2519(81,20)}$, $K_2 := \mathsf{KL25519(82,77)}$ and $K_3 := \mathsf{KL2663(260,139)}$ over the three primes $2^{251}-9$, $2^{255}-19$ and $2^{266}-3$ respectively. Implementations of scalar multiplications for all three Kummer lines using Intel intrinsics have been done and the code is publicly available. Timing results on the Skylake and the Haswell processors of Intel indicate that both fixed base and variable base scalar multiplications for K_1 and K_2 are faster than those achieved by Sandy2x, which is a highly optimised SIMD implementation in assembly of the well known Curve25519; for example, on Skylake, variable base scalar multiplication on K_1 is faster than Curve25519 by about 30%. On Skylake, both fixed base and variable base scalar multiplication for K_3 are faster than Sandy2x; whereas on Haswell, fixed base scalar multiplication for K_3 is faster than Sandy2x while variable base scalar multiplication for both K_3 and Sandy2x take roughly the same time. In fact, on Skylake, K_3 is both faster and also offers about 5 bits of higher security compared to Curve25519. In practical terms, the particular Kummer lines that are introduced in this work are serious candidates for deployment and standardisation. We further illustrate the usefulness of the proposed Kummer lines by instantiating the quotient Digital Signature Algorithm (qDSA) on all the three Kummer lines.
Keywords: elliptic curve cryptography, Kummer line, Montgomery curve, scalar multiplication.

1 Introduction

Curve-based cryptography provides a platform for secure and efficient implementation of public key schemes whose security rely on the hardness of discrete logarithm problem. Starting from the pioneering works of Koblitz [38] and Miller [42] introducing elliptic curves and the work of Koblitz [39] introducing hyperelliptic curves for cryptographic use, the last three decades have seen an extensive amount of research in the area.

Appropriately chosen elliptic curves and genus two hyperelliptic curves are considered to be suitable for practical implementation. Table 1 summarises features for some of the concrete curves that have been proposed in the literature. Arguably, the two most well known curves proposed till date for the 128-bit security level are P-256 [46] and Curve25519 [2]. Also the secp256k1 curve [51] has become very

---

* An earlier version of this work appeared as [37], and was recommended by the program chairs of the conference for invitation to the Journal of Cryptology.

** Part of the work was done while the author was a post-doctoral fellow at the Turing Laboratory of the Indian Statistical Institute.

popular due to its deployment in the Bitcoin protocol. All of these curves are in the setting of genus one over prime order fields. In particular, we note that Curve25519 has been extensively deployed for various applications. A listing of such applications can be found at [19]. So, from the point of view of deployment, practitioners are very familiar with genus one curves over prime order fields. Influential organisations, such as NIST, Brainpool, Microsoft (the NUMS curves) have concrete proposals in this setting. See [5] for a further listing of such primes and curves. It is quite likely that any future portfolio of proposals by standardisation bodies will include at least one curve in the setting of genus one over a prime field.

Table 1. Features of some curves proposed in the last few years.

Reference	genus	form	field order	endomorphisms
NIST P-256 [46]	1	Weierstrass	prime	no
Curve25519 [2]	1	Montgomery	prime	no
secp256k1 [51]	1	Weierstrass	prime	no
Brainpool [11]	1	Weierstrass	prime	no
NUMS [54]	1	twisted Edwards	prime	no
Longa-Sica [41]	1	twisted Edwards	p^2	yes
Bos et al. [10]	2	Kummer	p^2	yes
Hankerson et al. [34], Oliviera et al. [48]	1	Weierstrass/Koblitz	2^n	yes
Longa-Sica [41], Faz-Hernández et al. [21]	1	twisted Edwards	p^2	yes
Costello et al. [17]	1	Montgomery	p^2	yes
Gaudry-Schost [31], Bos et al. [9], Bernstein et al. [4]	2	Kummer	prime	no
Costello-Longa [16]	1	twisted Edwards	p^2	yes
Hankerson et al. [34], Oliviera et al. [49]	1	Weierstrass/Koblitz	2^n	yes
This work	1	Kummer	prime	no

Our Contributions

The contribution of this paper is to propose new curves for the setting of genus one over a prime order field. Actual scalar multiplication is done over the Kummer line associated with such a curve. The idea of using Kummer line was proposed by Gaudry and Lubicz [30]. They, however, were not clear about whether competitive speeds can be obtained using this approach. Our main contribution is to show that this can indeed be done using the single-instruction multiple-data (SIMD) instructions available in modern processors. We note that the use of SIMD instructions to speed up computation has been earlier proposed for Kummer surface associated with genus two hyperelliptic curves [30]. The application of this idea, however, to Kummer line has not been considered in the literature. Our work fills this gap and shows that properly using SIMD instructions provides a competitive alternative to known curves in the setting of genus one and prime order fields.

As in the case of Montgomery curve [43], scalar multiplication on the Kummer line proceeds via a laddering algorithm. A ladder step corresponds to each bit of the scalar and each such step consists of a doubling and a differential addition irrespective of the value of the bit. This makes the algorithm amenable to constant time implementation. We describe and implement a vectorised version of the laddering algorithm. This implementation is constant time in the sense that the time required to compute a scalar multiplication is almost the same irrespective of the value of the scalar.

Choice of the underlying field: Our target is the 128-bit security level. To this end, we consider three primes, namely, $2^{251} - 9$, $2^{255} - 19$ and $2^{266} - 3$. These primes are abbreviated as $p2519$, $p25519$ and $p2663$ respectively. The underlying field will be denoted as $\mathbb{F}_p$ where p is one of $p2519$, $p25519$ or $p2663$.

Choice of the Kummer line: Following previous suggestions [9, 3], we work in the square-only setting. In this case, the parameters of the Kummer line are given by two integers a^2 and b^2. We provide appropriate Kummer lines for all three of the primes $p2519$, $p25519$ and $p2663$. These are denoted as KL2519(81,20), KL25519(82,77) and KL2663(260,139) respectively. In each case, we identify a base point with small coordinates. The selection of the Kummer lines is done using a search for curves achieving certain desired security properties. Later we provide the details of these properties which indicate that the curves provide security at the 128-bit security level.

SIMD implementation: On Intel processors, it is possible to pack 4 64-bit words into a single 256-bit quantity and then use SIMD instructions to simultaneously work on the 4 64-bit words. We apply this approach to carefully consider various aspects of field arithmetic over $\mathbb{F}_p$. SIMD instructions allow the simultaneous computation of 4 multiplications in $\mathbb{F}_p$ and also 4 squarings in $\mathbb{F}_p$. The use of SIMD instructions dovetails very nicely with the scalar multiplication algorithm over the Kummer line as we explain below.

Scalar multiplication over the Kummer line: A uniform, ladder style algorithm is used. In terms of operation count, each ladder step requires 2 field multiplications, 6 field squarings, 6 multiplications by parameters and 2 multiplications by base point coordinates [30]. In contrast, the ladder step on the Montgomery curves requires 4 field multiplications, 4 squarings, 1 multiplication by curve parameter and 1 multiplication by a base point coordinate. This had led to Gaudry and Lubicz [30] commenting that Kummer line can be advantageous provided that the advantage of trading off multiplications for squarings is not offset by the extra multiplications by the parameters and the base point coordinates.

Our choices of the Kummer lines ensure that the parameters and the base point coordinates are indeed very small. This is not to suggest that the Kummer line is only suitable for fixed based point scalar multiplication. The main advantage arises from the structure of the ladder step on the Kummer line versus that on the Montgomery curve.

Following [2], an example of the ladder step on the Kummer line is shown in Figure 1. In this figure, the Hadamard transform $\mathcal{H}(u, v)$ is defined to be $(u + v, u - v)$. Observe that there are 4 layers of 4 simultaneous multiplications. The first layer consists of 2 field multiplications and 2 squarings, while the third layer consists of 4 field squarings. Using 256-bit SIMD instructions, the 2 multiplications and the 2 squarings in the first layer can be computed simultaneously using an implementation of vectorised field multiplication while the third layer can be computed using an implementation of vectorised field squaring. The second layer consists only of multiplications by parameters and is computed using an implementation of vectorised multiplication by constants. The fourth layer consists of two multiplications by parameters and two multiplications by base point coordinates. For fixed base point, this layer can be computed using a single vectorised multiplication by constants while for variable base point, this layer requires a vectorised field multiplication. A major advantage of the ladder step on the Kummer line is that the packing and unpacking into 256-bit quantities is done once each. Packing is done at the start of the scalar multiplication and unpacking is done at the end. The entire scalar multiplication can be computed on the packed vectorised quantities.

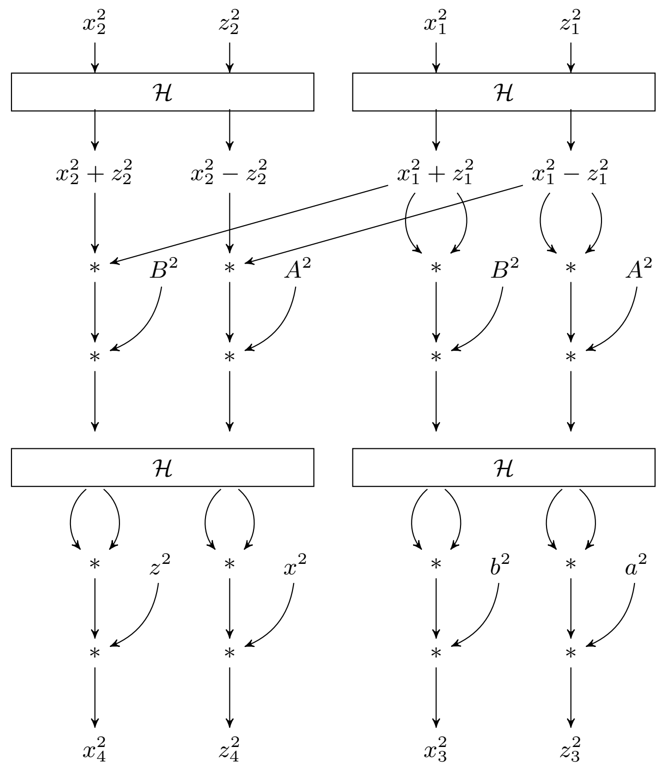

Fig. 1. One ladder step on the Kummer line.

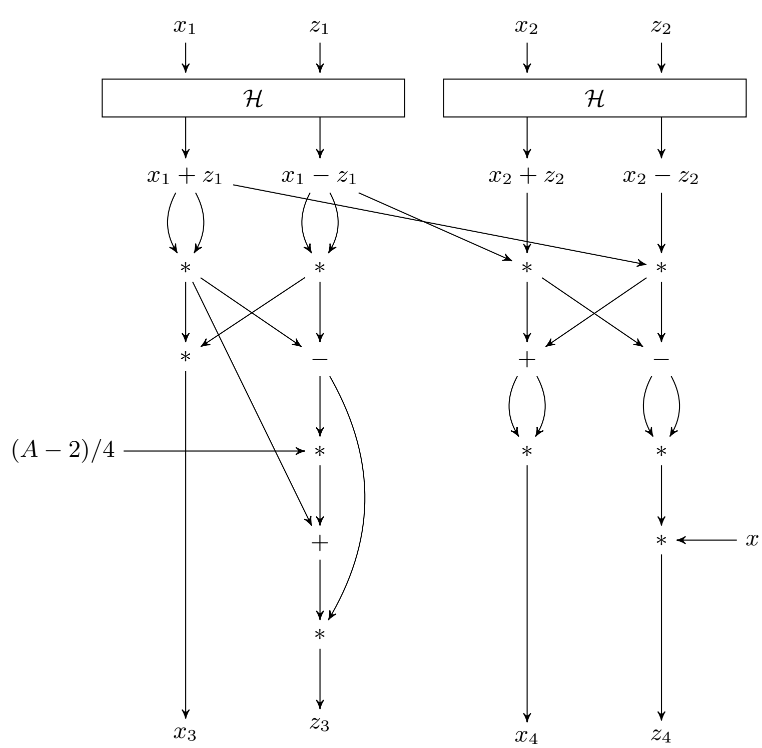

Fig. 2. One ladder step on the Montgomery curve. The figure has been reproduced from [2].

In contrast, the ladder step on the Montgomery curve is shown in Figure 2 which has been reproduced from [2]. The structure of this ladder is not as regular as the ladder step on the Kummer line. This makes it difficult to optimally group together the multiplications for SIMD implementation. Curve25519 is a Montgomery curve. SIMD implementations of Curve25519 have been reported in [18, 7, 13, 22]. The work [18] forms four groups of independent multiplications/squarings with the first and the third group consisting of four multiplications/squarings each, the second group consisting of two multiplications and the fourth group consists of a single multiplication. Interspersed with these multiplications are two groups each consisting of four independent additions/subtractions. The main problem with this approach is that of repeated packing/unpacking of data within a ladder step. This drawback will outweigh the benefits of four simultaneous SIMD multiplications and this approach has not been followed in later works [7, 13, 22]. These later implementations grouped together only two independent multiplications. In particular, we note that the well known Sandy2x implementation of Curve25519 is an SIMD implementation which is based on [13] and groups together only two multiplications. AVX2 based implementation of Curve25519 in [22] also groups together only 2 multiplications/squarings.

At a forum[3] Tung Chou comments (perhaps oblivious of [18]) that it would better to find four independent multiplications/squarings and vectorise them. As discussed above, the previous works on SIMD implementation of Curve25519 do not seem to have been able to identify this. On the other hand, for the ladder step on the Kummer line shown in Figure 1, performing vectorisation of 4 independent multiplications/squarings comes quite naturally. This indicates that the ladder step on the Kummer line is more SIMD friendly than the ladder step on the Montgomery curve.

Implementation: We report implementations of KL2519(81,20), KL25519(82,77) and KL2663(260,139). The implementations are in Intel intrinsics and use AVX2 instructions. On the Skylake processor, both fixed and variable base scalar multiplications for all the three Kummer lines are faster than Sandy2x which is the presently the best known SIMD implementation in assembly of Curve25519. On the earlier Haswell processor, both fixed and variable base scalar multiplications for KL2519(81,20), KL25519(82,77) are faster than that of Sandy2x; fixed base scalar multiplication for KL2663(260,139) is faster than that of Sandy2x while variable base scalar multiplication for both KL2663(260,139) and Sandy2x take roughly the same time. Detailed timing results are provided later.

At a broad level, the timing results reported in this work show that the availability of SIMD instructions leads to the following two practical consequences.

1. At the 128-bit security level, the choice of $\mathbb{F}_{2^{255}-19}$ as the base field is not the fastest. If one is willing to sacrifice about 2 bits of security, then using $\mathbb{F}_{2^{251}-9}$ as the base field leads to about 30% speed up on the Skylake processor.
2. More generally, the ladder step on the Kummer line is faster than the ladder step on the Montogomery curve. We have demonstrated this by implementing on the Intel processors. Future work can explore this issue on other platforms such as the ARM NEON architecture.

Instantiation of qDSA: The quotient digital signature algorithm (qDSA) was introduced by Renes and Smith [50]. It is defined on a pseudo-group using doubling and pseudo-addition operations which are combined using Montgomery's ladder. /Two instantiations of qDSA are given in [50]. The first is based on Curve25519 and the second is based on genus two Kummer surface. Further, subsequent to the conference version of the present paper, [35] has proposed the instantiation of qDSA on Kummer lines of the type considered in this work. To illustrate the usefulness of the Kummer lines introduced in this work, we provide concrete instantiations of qDSA on all the three Kummer lines. Complete implementation details are given and the code for the implementation has been made publicly available.

[3] `https://moderncrypto.org/mail-archive/curves/2015/000637.html`, accessed on September 1, 2018.

2 Background

In this section, we briefly describe theta functions over genus one, Kummer lines, Legendre form elliptic curves and their relations. In our description of the background material, we provide certain details which are not readily available in the literature. This is indicated at the relevant points.

2.1 Theta Functions

In this and the next few sections, we provide a sketch of the mathematical background on theta functions over genus one and Kummer lines. Following previous works [45, 36, 30], we define theta functions over the complex field. For cryptographic purposes, our goal is to work over a prime field of large characteristic. All the derivations that are used have a good reduction [30] and so it is possible to use the Lefschetz principle [1, 27] to carry over the identities proved over the complex to those over a large characteristic field.

Theta functions in genus one are called the Jacobi theta functions. For the general theory covering higher genus, we refer to [45, 36]. Cryptographic applications of theta functions were pointed out by Gaudry [28] for genus two and Gaudry and Lubicz [30] for genus one (and also for genus two over characteristic two fields). See also [45, 24, 23] for arithmetic on Kummer surface associated to genus two curves.

Let $\tau \in \mathbb{C}$ having a positive imaginary part and $w \in \mathbb{C}$. Let $\xi_1, \xi_2 \in \mathbb{Q}$. Theta functions with characteristics $\vartheta[\xi_1, \xi_2](w, \tau)$ are defined to be the following:

$$\vartheta[\xi_1, \xi_2](w, \tau) = \sum_{n \in \mathbb{Z}} \exp\left[\pi i (n + \xi_1)^2 \tau + 2\pi i (n + \xi_1)(w + \xi_2)\right]. \tag{1}$$

The scalars obtained by evaluating $\vartheta[\xi_1, \xi_2](w, \tau)$ at $w = 0$ are known as theta constants. We only consider the characteristics ξ_1 and ξ_2 which are in $\{0, \frac{1}{2}\}$ giving rise to four possible characteristics. Let $\xi_* = (-1)^{4\xi_1\xi_2}$. The relation between $\vartheta[\xi_1, \xi_2](w, \tau)$ and $\vartheta[\xi_1, \xi_2](-w, \tau)$ is the following. A proof is given in Appendix A.1.

$$\vartheta[\xi_1, \xi_2](-w, \tau) = \xi_* \cdot \vartheta[\xi_1, \xi_2](w, \tau). \tag{2}$$

Using this relation, the four characteristics can be divided into two groups. If $\xi_* = 1$, then the corresponding characteristic is said to be even and otherwise the characteristic is said to be odd. So, only the characteristic $[\frac{1}{2}, \frac{1}{2}]$ is odd and the other three are even.

For a fixed τ, the following theta functions are defined.

$$\vartheta_1(w) = \vartheta[0, 0](w, \tau) \quad \text{and} \quad \vartheta_2(w) = \vartheta\left[0, 1/2\right](w, \tau).$$
$$\Theta_1(w) = \vartheta[0, 0](w, 2\tau) \quad \text{and} \quad \Theta_2(w) = \vartheta\left[1/2, 0\right](w, 2\tau).$$

2.2 Theta Identities

The following identities hold for the theta functions. Proofs are given in Appendices A.2 and A.3.

$$\begin{aligned} 2\Theta_1(w_1 + w_2)\Theta_1(w_1 - w_2) &= \vartheta_1(w_1)\vartheta_1(w_2) + \vartheta_2(w_1)\vartheta_2(w_2); \\ 2\Theta_2(w_1 + w_2)\Theta_2(w_1 - w_2) &= \vartheta_1(w_1)\vartheta_1(w_2) - \vartheta_2(w_1)\vartheta_2(w_2); \end{aligned} \tag{3}$$

$$\begin{aligned} \vartheta_1(w_1 + w_2)\vartheta_1(w_1 - w_2) &= \Theta_1(2w_1)\Theta_1(2w_2) + \Theta_2(2w_1)\Theta_2(2w_2); \\ \vartheta_2(w_1 + w_2)\vartheta_2(w_1 - w_2) &= \Theta_1(2w_1)\Theta_1(2w_2) - \Theta_2(2w_1)\Theta_2(2w_2). \end{aligned} \tag{4}$$

Putting $w_1 = w_2 = w$, we obtain

$$\begin{aligned} 2\Theta_1(2w)\Theta_1(0) &= \vartheta_1(w)^2 + \vartheta_2(w)^2; \\ 2\Theta_2(2w)\Theta_2(0) &= \vartheta_1(w)^2 - \vartheta_2(w)^2; \end{aligned} \tag{5}$$

$$\begin{aligned} \vartheta_1(2w)\vartheta_1(0) &= \Theta_1(2w)^2 + \Theta_2(2w)^2; \\ \vartheta_2(2w)\vartheta_2(0) &= \Theta_1(2w)^2 - \Theta_2(2w)^2. \end{aligned} \tag{6}$$

Putting $w = 0$ in (5), we obtain

$$\begin{aligned} 2\Theta_1(0)^2 &= \vartheta_1(0)^2 + \vartheta_2(0)^2; \\ 2\Theta_2(0)^2 &= \vartheta_1(0)^2 - \vartheta_2(0)^2. \end{aligned} \tag{7}$$

2.3 Kummer Line

Let $\tau \in \mathbb{C}$ having a positive imaginary part and denote by $\mathbb{P}^1(\mathbb{C})$ the projective line over $\mathbb{C}$. The Kummer line $(\mathcal{K})$ associated with τ is the image of the map φ from $\mathbb{C}$ to $\mathbb{P}^1(\mathbb{C})$ defined by

$$\varphi : w \longmapsto [\vartheta_1(w) : \vartheta_2(w)]. \tag{8}$$

Suppose that $\varphi(w) = [\vartheta_1(w) : \vartheta_2(w)]$ is known for some $w \in \mathbb{F}_q$. Using (5) it is possible to compute $\Theta_1(2w)$ and $\Theta_2(2w)$ and then using (6) it is possible to compute $\vartheta_1(2w)$ and $\vartheta_2(2w)$. So, from $\varphi(w)$ it is possible to compute $\varphi(2w) = [\vartheta_1(2w) : \vartheta_2(2w)]$ without knowing the value of w.

Suppose that $\varphi(w_1) = [\vartheta_1(w_1) : \vartheta_2(w_1)]$ and $\varphi(w_2) = [\vartheta_1(w_2) : \vartheta_2(w_2)]$ are known for some $w_1, w_2 \in \mathbb{F}_q$. Using (5), it is possible to obtain $\Theta_1(2w_1), \Theta_1(2w_2), \Theta_2(2w_1)$ and $\Theta_2(2w_2)$. Then (4) allows the computation of $\vartheta_1(w_1+w_2)\vartheta_1(w_1-w_2)$ and $\vartheta_2(w_1+w_2)\vartheta_2(w_1-w_2)$. Further, if $\varphi(w_1 - w_2) = [\vartheta_1(w_1 - w_2) : \vartheta_2(w_1 - w_2)]$ is known, then it is possible to obtain $\varphi(w_1 + w_2) = [\vartheta_1(w_1 + w_2) : \vartheta_2(w_1 + w_2)]$ without knowing the values of w_1 and w_2.

The task of computing $\varphi(2w)$ from $\varphi(w)$ is called doubling and the task of computing $\varphi(w_1 + w_2)$ from $\varphi(w_1)$, $\varphi(w_2)$ and $\varphi(w_1 - w_2)$ is called differential (or pseudo) addition.

2.4 Square Only Setting

Let $P = \varphi(w) = [x : z]$ be a point on the Kummer line. As described above, doubling computes the point $2P$ and suppose that $2P = [x_3 : z_3]$. Further, suppose that instead of $[x : z]$, we have the values x^2 and z^2 and after the doubling we are interested in the values x_3^2 and z_3^2. Then the doubling operation only involves the squared quantities $\vartheta_1(0)^2, \vartheta_2(0)^2, \Theta_1(0)^2, \Theta_2(0)^2$ and x^2, z^2. As a consequence, the double of $[x : z]$ and $[x : -z]$ are same.

Similarly, consider that from $P_1 = \varphi(w_1) = [x_1 : z_1]$, $P_2 = \varphi(w_2) = [x_2 : z_2]$ and $P = P_1 - P_2 = \varphi(w_1 - w_2) = [x : z]$ the requirement is to compute $P_1 + P_2 = \varphi(w_1 + w_2) = [x_3 : z_3]$. If we have the values $x_1^2, z_1^2, x_2^2, z_2^2$ and x^2, z^2 along with $\vartheta_1(0)^2, \vartheta_2(0)^2, \Theta_1(0)^2, \Theta_2(0)^2$ then we can compute the values x_3^2 and z_3^2.

This approach requires only squared values, i.e., it starts with squared values and also returns squared values. Hence, this is called the square only setting. Note that in the square only setting, $[x^2 : z^2]$ represents two points $[x : \pm z]$ on the Kummer line[4]. For the case of genus two, the square only setting was advocated in [3, 9] (see also [15]).

Let

$$a^2 = \vartheta_1(0)^2,\ b^2 = \vartheta_2(0)^2,\ A^2 = a^2 + b^2 \text{ and } B^2 = a^2 - b^2.$$

[4] A reviewer has pointed out that explicit formulas appear at `https://hyperelliptic.org/EFD/g1p/auto-edwards-yzsquared.html#ladder-ladd-2006-g` (accessed on September 1, 2018).

Then from (7) we obtain $\Theta_1(0)^2 = A^2/2$ and $\Theta_2(0)^2 = B^2/2$. *By* $\mathcal{K}_{a^2,b^2}$ *we denote the Kummer line having the parameters* a^2 *and* b^2*.*

We provide the details of doubling in the square only setting. Using (5), we obtain

$$\Theta_1(2w)^2 = \frac{(x^2+z^2)^2}{2A^2};\ \Theta_2(2w)^2 = \frac{(x^2-z^2)^2}{2B^2}.$$

Then from (6)

$$x_3'^2 = \vartheta_1(2w)^2 = \frac{(\Theta_1(2w)^2+\Theta_2(2w)^2)^2}{a^2};\ z_3'^2 = \vartheta_2(2w)^2 = \frac{(\Theta_1(2w)^2-\Theta_2(2w)^2)^2}{b^2}.$$

For $[x:z] \in \mathbb{P}^1(\mathbb{C})$, $[x:z] = [\lambda x : \lambda z]$ for any non-zero λ. Suppose that we compute λx^2 and λz^2 for some non-zero λ. Since $\mathbb{C}$ is algebraically closed, there is a $\zeta \in \mathbb{C}$ such that $\zeta^2 = \lambda$ and so we in effect have $[\zeta^2 x^2 : \zeta^2 z^2]$. By taking square roots, we obtain the point $[\zeta x : \pm\zeta z] = [x : \pm z]$. So, in the square only setting, it is sufficient to compute $[\lambda x^2 : \lambda z^2]$ for some non-zero λ. Using this, we have

$$\begin{aligned}
[x_3'^2 : z_3'^2] &= \left[\frac{(\Theta_1(2w)^2+\Theta_2(2w)^2)^2}{a^2} : \frac{(\Theta_1(2w)^2-\Theta_2(2w)^2)^2}{b^2}\right] \\
&= \left[b^2(\Theta_1(2w)^2+\Theta_2(2w)^2)^2 : a^2(\Theta_1(2w)^2-\Theta_2(2w)^2)^2\right] \\
&= \left[b^2\left(\frac{(x^2+z^2)^2}{2A^2}+\frac{(x^2-z^2)^2}{2B^2}\right)^2 : a^2\left(\frac{(x^2+z^2)^2}{2A^2}-\frac{(x^2-z^2)^2}{2B^2}\right)^2\right] \\
&= \left[b^2\left(B^2(x^2+z^2)^2+A^2(x^2-z^2)^2\right)^2 : a^2\left(B^2(x^2+z^2)^2-A^2(x^2-z^2)^2\right)^2\right] \\
&= [x_3^2 : z_3^2].
\end{aligned}$$

So, it is sufficient to compute $[x_3^2 : z_3^2]$. The computation of $[x_3^2 : z_3^2]$ is shown as Algorithm dbl in Table 2. Note that $[x_3^2 : z_3^2]$ are the squared values of the double of $[x : \pm z]$ and is not the double of $[x^2 : z^2]$. Informally, however, we will say that $[x_3^2 : z_3^2]$ is the double of $[x^2 : z^2]$. Given $x_1^2, z_1^2, x_2^2, z_2^2, x^2$ and z^2,

$\mathsf{dbl}(x^2,z^2)$	$\mathsf{diffAdd}(x_1^2,z_1^2,x_2^2,z_2^2,x^2,z^2)$
$s_0 = B^2(x^2+z^2)^2$;	$s_0 = B^2(x_1^2+z_1^2)(x_2^2+z_2^2)$;
$t_0 = A^2(x^2-z^2)^2$;	$t_0 = A^2(x_1^2-z_1^2)(x_2^2-z_2^2)$;
$x_3^2 = b^2(s_0+t_0)^2$;	$x_3^2 = z^2(s_0+t_0)^2$;
$z_3^2 = a^2(s_0-t_0)^2$;	$z_3^2 = x^2(s_0-t_0)^2$;
return (x_3^2, z_3^2).	return (x_3^2, z_3^2).

Table 2. Double and differential addition in the square-only setting.

the computation of x_3^2 and z_3^2 is shown as Algorithm $\mathsf{diffAdd}$ in Table 2.

In $\mathcal{K}_{a^2,b^2}$, the point $[a^2 : b^2]$ (representing $[a : \pm b]$) in the square only setting acts as the identity element for the differential addtion. This is proved by showing that

$$\left.\begin{aligned}
&\mathsf{diffAdd}(x^2,z^2,a^2,b^2,x^2,z^2) = [x^2:z^2]; \\
&\mathsf{diffAdd}(x^2,z^2,x^2,z^2,a^2,b^2) = \mathsf{dbl}(x^2,z^2).
\end{aligned}\right\} \quad (9)$$

Also, the double of $[b^2 : a^2]$ (representing $[b : \pm a]$) is $[a^2 : b^2]$ so that $[b^2 : a^2]$ is a point of order two. This is proved by showing that

$$\mathsf{dbl}(b^2,a^2) = [a^2:b^2]. \quad (10)$$

The relations (9) and (10) are proved by simplifying the expressions arising in the computations of dbl and diffAdd. As part of the code package at [25], we provide a SAGE script which performs the symbolic verifications of these calculations.

In the rest of the paper, we will work in the square only setting over a Kummer line $\mathcal{K}_{a^2,b^2}$ for some values of the parameters a^2 and b^2.

2.5 Scalar Multiplication

Suppose $P = [x_1^2 : z_1^2]$ and n be a positive integer. We wish to compute $nP = [x_n^2 : z_n^2]$. The method for doing this is given by Algorithm scalarMult in Table 3 which essentially implements the Montgomery ladder [43]. Let the ℓ-bit binary expansion of n be $n = (1, n_{\ell-2}, \ldots, n_0)$. Algorithm scalarMult goes through $\ell - 1$ ladder steps. Each ladder step takes the squared coordinates of two points as input and provides as output the squared coordinates of two other points.

A conceptual description of a ladder step is given in Figure 1. Suppose the squared coordinates of the two input points to a ladder step are $[x_1^2 : z_1^2]$ and $[x_2^2 : z_2^2]$. Also assume that the double of the point $[x_1^2 : z_1^2]$, and addition of the points $[x_1^2 : z_1^2]$ and $[x_2^2 : z_2^2]$ are required to be performed. Then the ladder produces $[x_3^2 : z_3^2]$ and $[x_4^2 : z_4^2]$, where $[x_3^2 : z_3^2] = \mathsf{dbl}(x_1^2 : z_1^2)$ and $[x_4^2 : z_4^2] = \mathsf{diffAdd}(x_1^2, z_1^2, x_2^2, z_2^2, x^2, z^2)$.

scalarMult(P, n) input: $P \in \mathcal{K}_{a,b}$; ℓ-bit scalar $n = (1, n_{\ell-2}, \ldots, n_0)$; output: nP; set $R = P$ and $S = \mathsf{dbl}(P)$; for $i = \ell - 2, \ell - 3, \ldots, 0$ do $(R, S) = \mathsf{ladder}(R, S, n_i)$; return R.	ladder$(R, S, \mathfrak{b})$ if $(\mathfrak{b} = 0)$ $S = \mathsf{diffAdd}(R, S, P)$; $R = \mathsf{dbl}(R)$; else $R = \mathsf{diffAdd}(R, S, P)$; $S = \mathsf{dbl}(S)$; return (R, S).

Table 3. Scalar multiplication using a ladder.

The input to the first ladder step are the (squared) coordinates of $(P, 2P)$. Suppose, at the i-th iteration, the input to the ladder step corresponds to $(kP, (k+1)P)$. If $n_i = 0$, then the output consists of the (squared) coordinates of the points $(2kP, (2k+1)P)$ and if $n_i = 1$, then the output consists of the (squared) coordinates of $((2k+1)P, (2k+2)P)$.

2.6 Legendre Form Elliptic Curve

Let E be an elliptic curve and $\sigma : E \to E$ be the automorphism which maps a point of E to its inverse, i.e., for $(a, b) \in E$, $\sigma(a, b) = (a, -b)$.

For $\mu \in \mathbb{F}_q$, let

$$E_\mu : Y^2 = X(X-1)(X-\mu) \tag{11}$$

be an elliptic curve in the Legendre form. Let $\mathcal{K}_{a^2,b^2}$ be a Kummer line such that

$$\mu = \frac{a^4}{a^4 - b^4}. \tag{12}$$

An explicit map $\psi : \mathcal{K}_{a^2,b^2} \to E_\mu/\sigma$ has been given in [30]. In the square only setting, let $[x^2 : z^2]$ represent the points $[x : \pm z]$ of the Kummer line $\mathcal{K}_{a^2,b^2}$ such that $[x^2 : z^2] \neq [b^2 : a^2]$. Recall that

$[b^2 : a^2]$ has order two and $[a^2 : b^2]$ acts as the identity in $\mathcal{K}_{a^2,b^2}$. Then following [30], the map ψ in projective coordinates is the following.

$$\psi([x^2 : z^2]) = \begin{cases} \infty & \text{if } [x^2 : z^2] = [a^2 : b^2]; \\ \left[a^2x^2 : \cdot : a^2x^2 - b^2z^2\right] & \text{otherwise .} \end{cases} \tag{13}$$

Given $X = a^2x^2/(a^2x^2 - b^2z^2)$, it is possible to find $\pm Y$ from the equation of E, though it is not possible to uniquely determine the sign of Y. The inverse ψ^{-1} maps an element of E_μ/σ to the squared coordinates of points in $\mathcal{K}_{a^2,b^2}$. Let $\mathbf{P} = [X : \cdot : Z] \in E_\mu/\sigma$ be a point which is not of order two so that $X \neq 0, 1, \mu$. Then

$$\psi^{-1}(\mathbf{P}) = \begin{cases} [a^2 : b^2] & \text{if } \mathbf{P} = \infty; \\ \left[b^2X : a^2(X - Z)\right] & \text{if } \mathbf{P} = [X : \cdot : Z]. \end{cases} \tag{14}$$

Notation: We will use upper-case bold face letters such as $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{R}$ and $\mathbf{T}$ to denote points of E_μ and upper case normal letters such as P, Q and R to denote points of $\mathcal{K}_{a^2,b^2}$.

2.7 Consistency

Let $\mathcal{K}_{a^2,b^2}$ and E_μ be such that (12) holds. Consider the point $\mathbf{T} = [\mu : 0 : 1]$ on E_μ. Note that $\mathbf{T}$ is a point of order two. Given any point $\mathbf{P} = [X : \cdot : Z]$ of E_μ, let $\mathbf{Q} = \mathbf{P} + \mathbf{T}$. Then it is easy to verify that $\mathbf{Q} = [\mu(X - Z) : \cdot : X - \mu Z]$. Consider the map $\widehat{\psi} : \mathcal{K}_{a^2,b^2} \to E_\mu$ such that for points $[x : \pm z]$ represented by $[x^2 : z^2]$ in the square only setting

$$\widehat{\psi}([x^2 : z^2]) = \psi([x^2 : z^2]) + \mathbf{T}. \tag{15}$$

The inverse map $\widehat{\psi}^{-1}$ takes a point $\mathbf{P}$ of E_μ to squared coordinates in $\mathcal{K}_{a^2,b^2}$ and is given by

$$\widehat{\psi}^{-1}(\mathbf{P}) = \psi^{-1}(\mathbf{P} + \mathbf{T}). \tag{16}$$

(Since $\mathbf{T}$ is a point of order two, $\mathbf{T} = -\mathbf{T}$.)

For any points $\mathbf{P}_1, \mathbf{P}_2$ on E_μ which are not of order two and $\mathbf{P} = \mathbf{P}_1 - \mathbf{P}_2$, the following properties hold.

$$\left.\begin{aligned} 2 \cdot \widehat{\psi}([x^2 : z^2]) &= \widehat{\psi}(\mathsf{dbl}(x^2, z^2)); \\ \mathsf{dbl}\left(\widehat{\psi}^{-1}(\mathbf{P}_1)\right) &= \widehat{\psi}^{-1}\left(2\mathbf{P}_1\right); \\ \mathsf{diffAdd}\left(\widehat{\psi}^{-1}(\mathbf{P}_1), \widehat{\psi}^{-1}(\mathbf{P}_2), \widehat{\psi}^{-1}(\mathbf{P})\right) &= \widehat{\psi}^{-1}\left(\mathbf{P}_1 + \mathbf{P}_2\right). \end{aligned}\right\} \tag{17}$$

Note that $2 \cdot \widehat{\psi}([x^2 : z^2]) = 2 \cdot (\psi([x^2 : z^2]) + \mathbf{T}) = 2 \cdot \psi([x^2 : z^2])$. The proofs for (17) can be derived from the formulas for $\widehat{\psi}$, $\widehat{\psi}^{-1}$, the formulas for addition and doubling on E_μ, and the formulas arising from dbl and diffAdd. This involves simplifications of the intermediate expressions arising in these formulas. Such expressions become quite large. Instead of providing the calculations for simplifying these expressions, at [25] we provide a SAGE script which does the symbolic verification of the required calculations.

For the relation verifying the consistency of addition, the following point is to be noted. Suppose $\mathbf{P}_1 = (X_1, Y_1)$ and $\mathbf{P}_2 = (X_2, Y_2)$. Then $Y_1^2 = f(X_1)$ and $Y_2^2 = f(X_2)$ where $f(X) = X(X-1)(X-\mu)$. As a result, the expressions arising in the relation for verification of addition have to be reduced modulo $Y_1^2 - f(X_1)$ and $Y_2^2 - f(X_2)$ to obtain the desired equality. The SAGE script provided at [25] does this modulo reduction as part of the verification procedure.

The relations given by (17) have an important consequence to scalar multiplication. Suppose P is in $\mathcal{K}_{a^2,b^2}$ and $\mathbf{P} = \widehat{\psi}(P)$. Then $\widehat{\psi}(nP) = n\mathbf{P}$. Consequently, for P in $\mathcal{K}_{a^2,b^2}$ and $n \geq 0$, using $\widehat{\psi}(nP) = \psi(nP) + \mathbf{T}$ and $n\mathbf{P} = n\widehat{\psi}(P) = n(\psi(P) + \mathbf{T})$, we obtain

$$\psi(nP) = n\psi(P) + (n + 1 \bmod 2)\mathbf{T}. \tag{18}$$

Figure 3 depicts the relation $\widehat{\psi}(nP) = n\mathbf{P} = n\widehat{\psi}(P)$ in pictorial form.

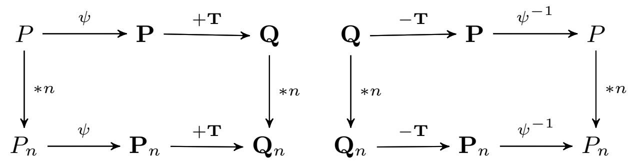

Fig. 3. Consistency of scalar multiplications on E_μ and $\mathcal{K}_{a^2,b^2}$.

2.8 Relation Between the Discrete Logarithm Problems

The equivalence of the hardness of solving the discrete log problem on a Legendre form elliptic curve and associated Kummer line is known and almost the same as a similar equivalence for genus 2. We refer to [52] and Section 5.4 of [29] for details. For the sake of completeness, we briefly mention this issue below.

Suppose the Kummer line $\mathcal{K}_{a^2,b^2}$ is chosen such that the corresponding curve E_μ has a cyclic subgroup $\mathfrak{G} = \langle \mathbf{P} \rangle$ of large prime order. Given $\mathbf{Q} \in \mathfrak{G}$, the discrete logarithm problem in $\mathfrak{G}$ is to obtain an n such that $\mathbf{Q} = n\mathbf{P}$. This problem can be reduced to computing discrete logarithm problem in $\mathcal{K}_{a^2,b^2}$. Map the point $\mathbf{P}$ (resp. $\mathbf{Q}$) to $P \in \mathcal{K}_{a,b}$ (resp. $Q \in \mathcal{K}_{a,b}$) using $\widehat{\psi}^{-1}$ Find n such that $Q = nP$ and return n. Similarly, the discrete logarithm problem in $\mathcal{K}_{a,b}$ can be reduced to the discrete logarithm problem in E_μ.

The above shows the equivalence of the hardness of solving the discrete logarithm problem in either E_μ or in $\mathcal{K}_{a^2,b^2}$. So, if E_μ is a well chosen curve such that the discrete logarithm problem in E_μ is conjectured to be hard, then the discrete logarithm problem in the associated $\mathcal{K}_{a^2,b^2}$ will be equally hard. This fact forms the basis for using Kummer line for cryptographic applications.

2.9 Recovering y-Coordinate

Suppose $\mathbf{Q} = (X_Q, Y_Q), \mathbf{R} = (X_R, Y_R), \mathbf{S} = (X_S, Y_S)$ are points in E_μ such that $\infty \neq \mathbf{Q} = \mathbf{R} - \mathbf{S}$, $\mathbf{Q} \neq \mathbf{R}$ and $\mathbf{Q}$ is not a point of order 2. The last two conditions imply that $X_Q \neq X_R$ and $Y_Q \neq 0$. So, it is allowed to divide by both $(X_R - X_Q)$ and Y_Q.

Suppose that X_Q, Y_Q, X_R and X_S are known. We show that Y_R is uniquely determined and can be computed from these four quantities. This is based on similar calculations in [43, **?**,47, 12]. For the genus two case, this problem has been addressed in [14].

Consider the chord-and-tangent rule for addition on E_μ. The points $\mathbf{R}$ and $-\mathbf{S}$ determine a line $Y = mX + c$. This line intersects the curve E_μ at the point $-\mathbf{Q} = (X_Q, -Y_Q)$. So, m can be determined as $m = (Y_R + Y_Q)/(X_R - X_Q)$. Substituting $Y = mX + c$ into the equation of the curve we obtain:

$$X^3 - (\mu + 1 + m^2)X^2 + (\mu - 2mc)X - c^2 = 0.$$

Since X_Q, X_R, X_S are roots of this equation, we have

$$X_Q + X_R + X_S = \mu + 1 + m^2.$$

Using the value for $m = (Y_R + Y_Q)/(X_R - X_Q)$, we have

$$(Y_R + Y_Q)^2 = (X_R - X_Q)^2(X_Q + X_R + X_S - \mu - 1).$$

Write $f(X) = X(X-1)(X-\mu)$. Then $Y_R^2 = f(X_R)$ and we obtain

$$Y_R = \frac{1}{2Y_Q}\left((X_R - X_Q)^2(X_Q + X_R + X_S - \mu - 1) - f(X_R) - Y_Q^2\right).$$

2.10 Scalar Multiplication in E_μ

Let E_μ be a Legendre form curve and $\mathcal{K}_{a^2,b^2}$ be a Kummer line in the square only setting. Suppose $\mathfrak{G} = \langle \mathbf{P} = (X_P, Y_P) \rangle$ is a cryptographically relevant subgroup of E_μ. Further, suppose a point $P = [x^2 : z^2]$ in $\mathcal{K}_{a^2,b^2}$ is known such that $(X_P, \ldots) = \widehat{\psi}(P) = \psi(P) + \mathbf{T}$ where as before $\mathbf{T} = (\mu, 0)$. The point P is the base point on $\mathcal{K}_{a^2,b^2}$ which corresponds to the point $\mathbf{P}$ on E_μ.

Let n be a non-negative integer which is less than the order of $\mathfrak{G}$. We show how to compute the scalar multiplication $n\mathbf{P}$ via the laddering algorithm on the Kummer line $\mathcal{K}_{a^2,b^2}$. First, the ladder algorithm is applied to the input P and n. This results in a pair of points Q and R, where $Q = nP$ and $R = (n+1)P$ so that $Q - R = -P$. The square only ladder algorithm will return $[x_Q^2 : z_Q^2]$ to represent Q and $[x_R^2 : z_R^2]$ to represent R. Let $\mathbf{Q} = \widehat{\psi}(Q) = \psi(Q) + \mathbf{T}$ and $\mathbf{R} = \widehat{\psi}(R) = \psi(R) + \mathbf{T}$. By the consistency of scalar multiplication, we have $\mathbf{Q} = n\mathbf{P}$.

Consider $\mathbf{Q} = \psi(Q) + \mathbf{T}$. Let $\alpha_Q = a^2 x_Q^2$ and $\beta_Q = a^2 x_Q^2 - b^2 z_Q^2$ so that $\psi(Q) = (\alpha_Q/\beta_Q, \ldots)$. Writing $\mathbf{Q} = (X_Q, Y_Q)$ and applying the addition rule on E_μ we obtain $X_Q = \gamma_Q/\delta_Q$ where $\gamma_Q = \mu(\alpha_Q - \beta_Q)$ and $\delta_Q = \alpha_Q - \mu\beta_Q$. Similarly, we obtain $\mathbf{R} = (X_R, \ldots)$ where $X_R = \gamma_R/\delta_R$ and γ_R, δ_R are given by the previous expression with Q replaced by R.

At this point, we have $\mathbf{P} = (X_P, Y_P)$, $\mathbf{Q} = (X_Q, Y_Q)$ and $\mathbf{R} = (X_R, \ldots)$ where $\mathbf{Q} - \mathbf{R} = -\mathbf{P}$. The y-coordinate Y_Q of $\mathbf{Q}$ can be recovered as discussed in Section 2.9. This gives

$$
\begin{aligned}
Y_Q &= \frac{-1}{2Y_P}\left((X_Q - X_P)^2(X_P + X_Q + X_R - \mu - 1) - f(X_Q) - Y_P^2\right) \\
&= -\frac{1}{2Y_P}\left(\left(\frac{\gamma_Q}{\delta_Q} - X_P\right)^2\left(X_P + \frac{\gamma_Q}{\delta_Q} + \frac{\gamma_R}{\delta_R} - \mu - 1\right) - f\left(\frac{\gamma_Q}{\delta_Q}\right) - Y_P^2\right).
\end{aligned}
$$

This shows that given n, it is possible to compute $\mathbf{Q} = (X_Q, Y_Q)$ such that $\mathbf{Q} = n\mathbf{P}$.

It is required to compute both Y_Q and X_Q. Using Montgomery's trick, the two inversions required for computing Y_Q and X_Q can be done using one inversion and 3 multiplications. So, the entire computation of X_Q and Y_Q from Q, R and P can be done using one inversion and a few multiplications in $\mathbb{F}_p$. The main time consuming step will be that of the inversion. If projective coordinates are used to represent the point of E_μ, then the field inversion can be avoided.

3 Kummer Line Over Prime Order Fields

Let p be a prime and $\mathbb{F}_p$ be the field of p elements. As mentioned earlier, using the Lefschetz principle, the theta identities also hold over $\mathbb{F}_p$. Consequently, it is possible to work over a Kummer line $\mathcal{K}_{a^2,b^2}$ and associated elliptic curve E_μ defined over the algebraic closure of $\mathbb{F}_p$. The only condition for this to be meaningful is that $a^4 - b^4 \neq 0 \bmod p$ so that $\mu = a^4/(a^4 - b^4)$ is defined over $\mathbb{F}_p$. The discriminant of E_μ is $\mu^2(\mu - 1)^2$ and so for the discriminant to be non-zero, the condition $ab \neq 0 \bmod p$ is required.

Following [3], we work in the square only setting. We choose a^2 and b^2 to be small values while p is a large prime and so the condition $a^4 - b^4 \neq 0 \bmod p$ easily holds. Note that we will choose a^2 and b^2 to be in $\mathbb{F}_p$ without necessarily requiring a and b themselves to be in $\mathbb{F}_p$. Similarly, in the square only setting when we work with squared representation $[x^2 : z^2]$ of points $[x : \pm z]$, the values x^2, z^2 will be in $\mathbb{F}_p$ and it is not necessary for x and z themselves to be in $\mathbb{F}_p$.

Our target is the 128-bit security level. To this end, we consider the three primes $p2519$, $p25519$ and $p2663$. The choice of these three primes is motivated by the consideration that these are of the form $2^m - \delta$, where m is around 256 and δ is a small positive integer. For m in the range 250 to 270 and $\delta < 20$, the only three primes of the form $2^m - \delta$ are $p2519$, $p25519$ and $p2663$. We later discuss the comparative advantages and disadvantages of using Kummer lines based on these three primes.

3.1 Finding a Secure Kummer Line

For each prime p, the procedure for finding a suitable Kummer line is the following. The value of a^2 is increased from 2 onwards and for each value of a^2, the value of b^2 is varied from 1 to $a^2 - 1$; for each pair (a^2, b^2), the value of $\mu = a^4/(a^4 - b^4)$ is computed and the order of $E_\mu(\mathbb{F}_p)$ is computed. Let $t = p + 1 - \#E_\mu(\mathbb{F}_p)$. Let ℓ and ℓ_T be the largest prime factors of $p + 1 - t$ and $p + 1 + t$ respectively and let $h = (p+1-t)/\ell$ and $h_T = (p+1+t)/\ell_T$. Here h and h_T are the co-factors of the curve and its quadratic twists respectively. If both h and h_T are small, then (a^2, b^2) is considered. Among the possible (a^2, b^2) that were obtained, we have used the one with the minimum value of a^2. After fixing (a^2, b^2) the following parameters for E_μ have been computed.

1. Embedding degrees k and k_T of the curve and its twist. Here k (resp. k_T) is the smallest positive integer such that $\ell | p^k - 1$ (resp. $\ell_T | p^{k_T} - 1$). This is given by the order of p in $\mathbb{F}_\ell$ (resp. $\mathbb{F}_{\ell_T}$) and is found by checking the factors of $\ell - 1$ (resp. $\ell_T - 1$).
2. The complex multiplication field discriminant D. This is defined in the following manner[5]: By Hasse's theorem, $|t| \leq 2\sqrt{p}$ and in the cases that we considered $|t| < 2\sqrt{p}$ so that $t^2 - 4p$ is a negative integer; let s^2 be the largest square dividing $t^2 - 4p$; define $D = (t^2 - 4p)/s^2$ if $t^2 - 4p \bmod 4 = 1$ and $D = 4(t^2 - 4p)/s^2$ otherwise. (Note that D is different from the discriminant of E_μ which is equal to $\mu^4 - 2\mu^3 + \mu^2$.)

Table 4 provides the three Kummer lines and (estimates of) the sizes of the various parameters of the associated Legendre form elliptic curves. As part of [25], we provide Magma code for computing these parameters and also their exact values. The Kummer line $\mathcal{K}_{a^2,b^2}$ over $p2519$ is compactly denoted as KL2519(a^2, b^2) and similarly for Kummer lines over $p25519$ and $p2663$. For each Kummer line reported in Table 4, the base point $[x^2 : z^2]$ is such that its order is ℓ. Table 4 also provides the corresponding details for Curve25519, P-256 and secp256k1 which have been collected from [5]. This will help in comparing the new proposals with some of the most important and widely used proposals over prime fields that are present in the literature.

Table 4. New Kummer lines and their parameters in comparison to Curve25519, P-256 and secp256k1.

	KL2519$(81, 20)$	KL25519$(82, 77)$	KL2663$(260, 139)$	Curve25519 [2]	P-256 [46]	secp256k1 [51]
$(\lg \ell, \lg \ell_T)$	$(248, 248)$	$(251.4, 252)$	$(262.4, 263)$	$(252, 253)$	$(256, 240)$	$(256, 219.3)$
(h, h_T)	$(8, 8)$	$(12, 8)$	$(12, 8)$	$(8, 4)$	$(1, 3 \cdot 5 \cdot 13 \cdot 179)$	$(1, 3^2 \cdot 13^2 \cdot 3319 \cdot 22639)$
(k, k_T)	$\left(\ell - 1, \frac{\ell_T - 1}{7}\right)$	$(\ell - 1, \ell_T - 1)$	$\left(\frac{\ell-1}{2}, \ell_T - 1\right)$	$\left(\frac{\ell-1}{6}, \ell_T - 1\right)$	$\left(\frac{\ell-1}{3}, \frac{\ell_T-1}{2}\right)$	$\left(\frac{\ell-1}{6}, \frac{\ell_T-1}{6}\right)$
$\lg(-D)$	246.3	255	266	254.7	258	1.58
base point	$[64 : 1]$	$[31 : 1]$	$[2 : 1]$	$(9, \ldots)$	large	large

The Four-$\mathbb{Q}$ proposal [16] is an elliptic curve over $\mathbb{F}_{p^2}$ where $p = 2^{127} - 1$. For this curve, the size ℓ of the cryptographic sub-group is 246 bits, the co-factor is 392 and the embedding degree is $(\ell - 1)/2$. The largest prime dividing the twist order is 158 bits and [16] does not consider twist security to be an issue.

For KL2519$(81, 20)$, $[15 : 1]$ is another choice of base point. Also, for $p2519$, KL2519$(101, 61)$ is another good choice for which both h and h_T are 8, the other security parameters have large values and $[4 : 1]$ is a base point. We have implementations of both KL2519$(81, 20)$ and KL2519$(101, 61)$ and the performances of both are almost the same. Hence, we report only the performance of KL2519$(81, 20)$.

[5] `https://safecurves.cr.yp.to/disc.html`, accessed on September 1, 2018.

The points of order two on the Legendre form curve $Y^2 = X(X-1)(X-\mu)$ are $(0,0),(1,0)$ and $(\mu,0)$. The sum of two distinct points of order two is also a point of order two and hence the sum is the third point of order two; as a result, the points of order two along with the identity form an order 4 subgroup of the group formed by the $\mathbb{F}_p$ rational points on the curve. Consequently, the group of $\mathbb{F}_p$ rational points has an order which is necessarily a multiple of 4, i.e., $p+1-t=4a$ for some integer a.

1. If $p=4m+1$, then $p+1+t=4a_T$ where $a_T=2m-a+1 \not\equiv a \bmod 2$. As a result, it is not possible to have both h and h_T to be equal to 4, or both of these to be equal to 8. So, the best possibilities for h and h_T are that one of them is 4 and the other is 8. The primes $p2519$ and $p2663$ are both $\equiv 1 \bmod 4$. For these two primes, searching for a^2 up to 512, we were unable to find any choice for which one of h and h_T is 4 and the other is 8. The next best possibilities for h and h_T are that one of them is 8 and the other is 12. We have indeed found such choices which are reported in Table 4.
2. If $p=4m+3$, then $p+1+t=4a_T$ where $a_T=2m-a+2 \equiv a \bmod 2$. In this case, it is possible that both h and h_T are equal to 4. The prime $p2519$ is $\equiv 1 \bmod 3$. For this prime, searching for a^2 up to 512, we were unable to find any choice where $h=h_T=4$. The next best possibility is $h=h_T=8$ and we have indeed found such a choice which is reported in Table 4.

Gaudry and Lubicz [30] had remarked that for Legendre form curves, if $p \equiv 1 \bmod 4$, then the orders of the curve and its twist are divisible by 4 and 8 respectively; while if $p \equiv 3 \bmod 4$, then the orders of the curve and its twist are divisible by 8 and 16 respectively. The Legendre form curve corresponding to $\mathsf{KL2519}(81,20)$ has $h=h_T=8$ and hence shows that the second statement is incorrect. The discussion provided above clarifies the issue of divisibility by 4 of the order of the curve and its twist.

Set of scalars: In Table 5, we define the permitted sets of scalars for the proposed Kummer lines. For comparison, we also provide the set of scalars for Curve25519.

Table 5. Allowed sets of scalars. In the table, $\mathfrak{s}$ is the number of available bits in the corresponding scalar.

Curve	set of scalars	$\mathfrak{s}$
Curve25519	$8(2^{251}+\{0,1,\ldots,2^{251}-1\})$	251
KL2519$(81,20)$	$8(2^{247}+\{0,1,\ldots,2^{247}-1\})$	247
KL25519$(82,77)$	$12(2^{252}+\{0,1,\ldots,2^{250}-1\})$	250
KL2663$(260,139)$	$12(2^{260}+\{0,1,\ldots,2^{258}-1\})$	258

The co-factors of both KL2519$(81,20)$ and Curve25519 are equal to 8. So, the set of scalars for KL2519$(81,20)$ is defined in a manner similar to that of Curve25519. For $x \in \{0,1,\ldots,2^{247}-1\}$, $8(2^{247}+x)$ is a 251-bit (32-byte) quantity having 1 in position numbered 250 (and 0 in positions numbered 251 to 255).

Consider the set of scalars for KL25519$(82,77)$. Let $x \in \{0,1,\ldots,2^{250}-1\}$ so that x is a 250-bit quantity spanning bit positions 0 to 249. Then $12x = 8x+4x < 2^{253}+2^{252} < 2^{254}$ and so $12x$ is a 254-bit quantity spanning bit positions 0 to 253. Also, $12 \cdot 2^{252} = 2^{255}+2^{254}$. Therefore, $12(2^{252}+x)$ is a 256-bit (32-byte) quantity having 1 in positions numbered 254 and 255. To ensure that the scalar is a multiple of the co-factor, it is sufficient to define the set of scalars to be $12\{0,1,\ldots,2^{250}-1\}$. The bit size of the resulting scalars, however, is not fixed and could be either 253 or 254 bits. This would result in the number of iterations of the Montgomery ladder not being fixed. Our definition of the set of scalars, on the other hand, result in all scalars being 256-bit long.

Similar to the case of KL25519$(82,77)$, for KL2663$(260,139)$ it is possible to argue that for $x \in \{0,1,\ldots,2^{258}-1\}$, $12(2^{260}+x)$ is a 264-bit (33-byte) quantity having 1 in positions numbered 262 and

263. It is possible to define a slightly larger set of scalars for KL2663(260, 139), namely, one can consider the set of scalars to be $12(2^{263} + \{0, 1, \ldots, 2^{261} - 1\})$. This, however, results in a 34-byte quantity. We have chosen the smaller set of scalars to ensure that a scalar fits within 33 bytes. Alternatively, one may define the set of scalars to be $12(2^{252} + \{0, 1, \ldots, 2^{250} - 1\})$ (which is same as the set of scalars defined for KL25519(82, 77)) so that a scalar becomes a 32-byte quantity. This would make a sub-optimal use of the 262-bit group size and would be desirable only if fitting into 32 bytes is an overriding concern.

We note the following consequences of the definitions of the sets of scalars.

Resistance to small subgroup attacks: The effectiveness of small subgroup attacks [40] is determined by the size of the co-factor. Such attacks can be prevented by multiplying the scalar by the co-factor h. The definition of the allowed sets of scalars in Table 5 ensures that the allowed scalars are multiples of the co-factor which in turn ensures resistance to small subgroup attacks.

Constant number of iterations: The number of iterations of the Montgomery ladder does not depend on the actual value of the scalar. For KL2519(81, 20), 251 pseudo-additions and 251 doublings are required; for KL25519(82, 77), 255 pseudo-additions and 255 doublings are required; and for KL2663(260, 139), 263 pseudo-additions and 263 doublings are required.

Clamping: For Curve25519, the procedure of formatting a 256-bit random string into a scalar of the allowed form has been called clamping[6]. Considering the 256-bit string to be a 32-byte quantity, clamping consists of clearing bits 0, 1 and 2 of the first byte, clearing bit 7 of the last byte, and setting bit 6 of the last byte (see Section 3 of [2]). This ensures that the resulting scalar is of the appropriate form for Curve25519 as shown in Table 5. We define the clamping procedures for the new curves.

Clamping for KL2519(81, 20): Since the co-factor of KL2519(81, 20) is 8, a clamping procedure similar to that of Curve25519 can be defined for KL2519(81, 20). Given a 32-byte quantity, clear bits 0, 1 and 2 of the first byte, set bit number 2 of the last byte and clear bits numbered 3 to 7 of the last byte.

Clamping for KL25519(82, 77): Given a 32-byte quantity, set bit number 4 of the last byte and clear bits numbered 2, 3 and 5 to 7 of the last byte. Multiply the resulting quantity by 12.

Clamping for KL2663(260, 139): Given a 33-byte quantity, set bit number 4 of the last byte and clear bits numbered 2, 3 and 5 to 7 of the last byte. Multiply the resulting quantity by 12.

Given a scalar v, by $\mathsf{clamp}(v)$, we will denote the clamping procedure applied to v. For KL2519(81, 20) and KL25519(82, 77), v is a 32-byte quantity while for KL2663(260, 139), v is a 33-byte quantity. The output of $\mathsf{clamp}(v)$ is a scalar in an appropriate form as shown in Table 5. Further, the clamping procedure ensures that if v is random, then $\mathsf{clamp}(v)$ has $\mathfrak{s}$ bits of randomness.

Twist security: Let $\mathfrak{r}$ be a quadratic non-residue in $\mathbb{F}_p$ and consider the curve $\mathfrak{r}Y^2 = f(X) = X(X - 1)(X - \mu)$. This is a quadratic twist of the original curve. For any $X \in \mathbb{F}_p$, either $f(X)$ is a quadratic residue or a quadratic non-residue. If $f(X)$ is a quadratic residue, then $(X, \pm\sqrt{f(X)})$ are points on the original curve; otherwise, $(X, \pm\sqrt{\mathfrak{r}^{-1}f(X)})$ are points on the quadratic twist. So, for each point X, there is a pair of points on the curve or on the quadratic twist. An x-coordinate only scalar multiplication algorithm does not distinguish between these two cases. One way to handle the problem is to check whether $f(X)$ is a quadratic residue before performing the scalar multiplication. This, however, has a significant cost. On the other hand, if this is not done, then an attacker may gain knowledge about the secret scalar modulo the co-factor of the twist. The twist co-factors of the new curves in Table 4 are all 8 which is only a little larger than the twist co-factor of 4 for Curve25519. Consequently, as in the case of Curve25519, attacks based on the co-factors of the twist are ineffective.

[6] https://cr.yp.to/ecdh.html, accessed on September 1, 2018

To summarise, at approximately the 128-bit security level, the security provided by the three new curves listed in Table 4 is comparable to other well known curves in the literature.

4 Field Arithmetic

As mentioned earlier, we consider three primes $p2519 = 2^{251} - 9$, $p25519 = 2^{255} - 19$ and $p2663 = 2^{266} - 3$. The general form of these primes is $p = 2^m - \delta$. Let η and ν be such that $m = \eta(\kappa - 1) + \nu$ with $0 \leq \nu < \eta$. The values of m, δ, κ, η and ν for $p2519$, $p25519$ and $p2663$ are given in Table 6. The value of κ indicates the number of limbs used to represent elements of $\mathbb{F}_p$; the value of η represents the number of bits in the first $\kappa - 1$ limbs; and the value of ν is the number of bits in the last limb. For each prime, two sets of values of κ, η and ν are provided. This indicates that two different representations of each prime will be used.

1. For the representations with $\kappa = 5$, each limb will fit into a 64-bit word and a field multiplication can be computed using several $64 \times 64 \rightarrow 128$ multiplications without any SIMD operations. The $\kappa = 5$ representations are used for computing the inversion which is required at the end for converting from projective to affine.
2. A Kummer line allows the execution of four simultaneous multiplications (and four simultaneous squarings). This can be computed using SIMD instructions (specifically, the AVX2 instructions on modern Intel processors). To avail such instructions, the limbs of four field elements are stored in one 256-bit word. A single SIMD multiplication performs four simultaneous $32 \times 32 \rightarrow 64$ multiplications. In this case, the representations of field elements with $\kappa = 9$ or $\kappa = 10$ are used.

The scalar multiplication on the Kummer line will be computed entirely using SIMD instructions. At the end, the result in projective coordinates is converted to affine coordinates using an inversion followed by a multiplication. The entire scalar multiplication is done using the longer representation (i.e., with $\kappa = 9$ or $\kappa = 10$); next the two components of the result are converted to the shorter representation (i.e., with $\kappa = 5$); and then the inversion and the single field multiplication are done using the representation with $\kappa = 5$.

Table 6. The different values of κ, η and ν corresponding to the primes $p2519$, $p25519$ and $p2663$.

prime	m	δ	κ	η	ν
$p2519$	251	9	9	28	27
			5	51	47
$p25519$	255	19	10	26	21
			5	51	51
$p2663$	266	3	10	27	23
			5	54	50

In the following sections, we describe methods to perform arithmetic over $\mathbb{F}_p$. Most of the description is in general terms of κ, η and ν. The specific values of κ, η and ν are required only to determine that no overflow occurs.

4.1 Representations of Field Elements

Let $\theta = 2^\eta$ and consider the polynomial $A(\theta)$ defined in the following manner:

$$A(\theta) = a_0 + a_1\theta + \cdots + a_{\kappa-1}\theta^{\kappa-1} \tag{19}$$

where $0 \leq a_0, \ldots, a_{\kappa-1} < 2^\eta$ and $0 \leq a_{\kappa-1} < 2^\nu$. Such a polynomial will be called a *proper* polynomial.

Note that proper polynomials are in 1-1 correspondence with the integers $0, \ldots, 2^m - 1$. This leads to non-unique representation of some elements of $\mathbb{F}_p$: specifically, the elements $0, \ldots, \delta - 1$ are also represented as $2^m - \delta, \ldots, 2^m - 1$. This, however, does not cause any of the computations to become incorrect. Conversion to unique representation using a simple constant time code is done once at the end of the computation. Suppose $A(\theta) = \sum_{i=0}^{\kappa-1} a_i\theta^i$ with $0 \leq a_i < 2^\eta$ is to be converted to a unique representation. The idea is the following.

- Initialise two arrays t_0 and t_1 of length κ as follows: $t_0[i] = a_i$ for $i = 0, \ldots, \kappa - 1$; and $t_1[i] = 0$, for $i = 1, \ldots, \kappa - 1$; $t_1[0] = a_0 - (2^\eta - \delta)$.
- Let $b = (b_{\kappa-1}, \ldots, b_0)$ where each b_i is a bit; $b_{\kappa-1} = 0$ if and only if $a_{\kappa-1} = 2^\nu - 1$ for $i = 1, \ldots, \kappa - 2$, $b_i = 0$ if and only if $a_i = 2^\eta - 1$; and $b_0 = 0$ if and only if $a_0 > 2^\eta - \delta - 1$.
- If $b = 0^\kappa$ then set $a_i = t_1[i]$ for $i = 0, \ldots, \kappa - 1$; else set $a_i = t_0[i]$ for $i = 0, \ldots, \kappa - 1$.

The procedure to reduce to the unique representation is made after the final inversion. As mentioned above, for computing the inversion, the representation with $\kappa = 5$ is used. So the bit array b in the above description can be conveniently represented using a byte (an unsigned char) and the bits of b are set to 0 as described. The check $b = 0^\kappa$ is simply a check whether the value of b is 0 or not. Since the procedure to create a unique representation is done once in the entire computation, its effect on the overall efficiency of scalar multiplication is insignificant.

We note that the issue of non-unique representation was already mentioned in [2] where the following was noted: ‘Note that integers are not converted to a unique “smallest” representation until the end of the Curve25519 computation. Producing reduced representations is generally much faster than producing “smallest” representations.’

4.2 Representation of the Prime p

The representation of the prime p will be denoted by $\mathfrak{P}(\theta)$ where

$$\begin{aligned} \mathfrak{P}(\theta) &= \textstyle\sum_{i=0}^{\kappa-1} \mathfrak{p}_i\theta^i \text{ with} \\ \mathfrak{p}_0 &= 2^\eta - \delta; \\ \mathfrak{p}_i &= 2^\eta - 1; \qquad i = 1, \ldots, \kappa - 2; \text{ and} \\ \mathfrak{p}_{\kappa-1} &= 2^\nu - 1. \end{aligned} \tag{20}$$

This representation will only be required for the larger value of κ.

4.3 Reduction

This operation will be required for both values of κ.

Using $p = 2^m - \delta$, for $i \geq 0$, we have $2^{m+i} = 2^i \times 2^m = 2^i(2^m - \delta) + 2^i\delta \equiv 2^i\delta \bmod p$. So, multiplying by 2^{m+i} modulo p is the same as multiplying by $2^i\delta$ modulo p. Recall that we have set $\theta = 2^\eta$ and so $\theta^\kappa = 2^{\eta\kappa} = 2^{m+\eta-\nu}$ which implies that

$$\theta^\kappa \bmod p = 2^{\eta-\nu}\delta. \tag{21}$$

Suppose $C(\theta) = \sum_{i=0}^{\kappa-1} c_i\theta^i$ is a polynomial such that for some $\mathfrak{m} \leq 64$, $c_i < 2^{\mathfrak{m}}$ for all $i = 0, \ldots, 7$. If for some $i \in \{0, \ldots, \kappa - 2\}$, $c_i \geq 2^\eta$, or $c_{\kappa-1} \geq 2^\nu$, then $C(\theta)$ is not a proper polynomial. Following the idea in [2, 7, 13], Table 7 describes a method to obtain a polynomial $D(\theta) = \sum_{i=0}^{\kappa-1} d_i\theta^i$ such that $D(\theta) \equiv C(\theta) \bmod p$. For $i = 0, \ldots, \kappa - 2$, Step 3 ensures $c_i + s_i = d_i + 2^\eta s_{i+1}$ and $d_i < 2^\eta$; Step 5 ensures

$\mathsf{reduce}(C(\theta))$
input: $C(\theta) = c_0 + c_1\theta + \cdots + c_{\kappa-1}\theta^{\kappa-1}$, $c_i < 2^{\mathfrak{m}}$, $i = 0, \ldots, \kappa-1$;
output: polynomial $D(\theta)$ such that $D(\theta) \equiv C(\theta) \bmod p$;
1. $s_0 \leftarrow 0$;
2. for $i = 0, \ldots, \kappa-2$ do
3. $d_i \leftarrow \mathsf{lsb}_\eta(c_i + s_i)$; $s_{i+1} \leftarrow (c_i + s_i)/2^\eta$;
4. end for;
5. $d_{\kappa-1} \leftarrow \mathsf{lsb}_\nu(c_{\kappa-1} + s_{\kappa-1})$; $t_0 = (c_{\kappa-1} + s_{\kappa-1})/2^\nu$;
6. $e_0 \leftarrow \mathsf{lsb}_\eta(d_0 + 2^{\eta-\nu}\delta t_0)$; $t_1 \leftarrow (d_0 + 2^{\eta-\nu}\delta t_0)/2^\eta$; [$t_2 \leftarrow \lfloor (d_1 + t_1)/2^\eta \rfloor$]
7. $d_0 \leftarrow e_0$; $d_1 \leftarrow d_1 + t_1$;
8. return $D(\theta)$.

Table 7. The reduction algorithm.

$c_{\kappa-1} + s_{\kappa-1} = d_{\kappa-1} + 2^\nu t_0$ and $d_{\kappa-1} < 2^\nu$. In Step 6, t_2 is actually not computed, it is provided for the ease of analysis.

Using $\theta = 2^\eta$, the computation can be written out more explicitly in the following manner.

$$\begin{aligned}
C(\theta) &= c_0 + c_1\theta + \cdots + c_{\kappa-1}\theta^{\kappa-1} \\
&= (d_0 + s_1\theta) + c_1\theta + \cdots + c_{\kappa-1}\theta^{\kappa-1} \\
&= d_0 + (s_1 + c_1)\theta + \cdots + c_{\kappa-1}\theta^{\kappa-1} \\
&= d_0 + ((d_1 + s_2\theta))\theta + \cdots + c_{\kappa-1}\theta^{\kappa-1} \\
&= d_0 + d_1\theta + (s_2 + c_2)\theta^2 + \cdots + c_{\kappa-1}\theta^{\kappa-1} \\
&\ \cdot\ \cdots \\
&= d_0 + d_1\theta + d_2\theta^2 + \cdots + (s_{\kappa-1} + c_{\kappa-1})\theta^{\kappa-1} \\
&= d_0 + d_1\theta + d_2\theta^2 + \cdots + (d_{\kappa-1} + t_0\theta)\theta^{\kappa-1} \\
&= d_0 + d_1\theta + d_2\theta^2 + \cdots + d_{\kappa-1}\theta^{\kappa-1} + t_0\theta^\kappa \\
&\equiv (d_0 + 2^{\eta-\nu}\delta t_0) + d_1\theta + d_2\theta^2 + \cdots + d_{\kappa-1}\theta^{\kappa-1} \pmod p \\
&= (e_0 + t_1\theta) + d_1\theta + d_2\theta^2 + \cdots + d_{\kappa-1}\theta^{\kappa-1} \\
&= e_0 + (d_1 + t_1)\theta + d_2\theta^2 + \cdots + d_{\kappa-1}\theta^{\kappa-1} \\
&= D(\theta).
\end{aligned}$$

We have used the fact that $\theta^\kappa \equiv 2^{\eta-\nu}\delta \bmod p$. This computation shows that $D(\theta) \equiv C(\theta) \bmod p$.

For $i = 1, \ldots, \kappa-1$, let $\mathfrak{b}_i$ be the maximum value that s_i can take; let $\mathfrak{d}_0$ and $\mathfrak{d}_1$ respectively be the maximum values that t_0 and t_1 can take. Then

- $\mathfrak{b}_1 = 2^{\mathfrak{m}-\eta}$;
- $\mathfrak{b}_i = \lfloor (2^{\mathfrak{m}} + \mathfrak{b}_{i-1})/2^\eta \rfloor$, for $i = 2, \ldots, \kappa-1$;
- $\mathfrak{d}_0 = \lfloor (2^{\mathfrak{m}} + \mathfrak{b}_{\kappa-1})/2^\nu \rfloor$;
- $\mathfrak{d}_1 = \lfloor (2^\eta - 1 + 2^{\eta-\nu}\delta\mathfrak{d}_0)/2^\eta \rfloor$.

The values of $\mathfrak{b}_i$, $\mathfrak{d}_0$ and $\mathfrak{d}_1$ are determined entirely by $\mathfrak{m}$, η, ν and δ. The maximum possible value of $\mathfrak{m}$ is 64 and the values of η and ν are determined by the choice of the prime p. For each of the primes $p2519$, $p25519$ and $p2663$, it turns out that $\mathfrak{d}_1 < 2^\eta - 1$. Since $d_1 \leq 2^\eta - 1$, $t_2 \leq 1$ and so the updated value of d_1 at Step 7 is less than $2^{\eta+1} - 2$. So, if $t_2 = 1$, then $\mathsf{lsb}_\eta(d_1)$ is less than $2^\eta - 1$. So, $D(\theta)$ is not necessarily a proper polynomial as the bound on d_1 can possibly be violated, though the bounds on all the other d_i's hold.

We first argue that $\mathsf{reduce}(D(\theta))$ is indeed a proper polynomial. Suppose $\mathsf{reduce}(D(\theta))$ returns $D'(\theta) = \sum_{i=0}^{\kappa-1} d'_i\theta^i$. The values of $d'_0, \ldots, d'_{\kappa-1}$ are computed by the reduce algorithm with d'_0 and

d'_1 being first computed in Step 3 and then updated in Step 7. Let $s'_1, \ldots, s'_{\kappa-1}, t'_0, t'_1$ be the values of the s and t variables when reduce is applied to $D(\theta)$. Since $d_0 < 2^\eta$, $s'_1 = 0$ and we have $s'_2 = t_2$. It is now easy to argue that the corresponding $s'_3, \ldots, s'_{\kappa-1}, t'_0, t'_1$ are all at most 1. We have $d'_0 = d_0$ and $d'_1 = \mathsf{lsb}_\eta(d_1)$ at Step 3. If $t_2 = 1$, then $d'_1 < 2^\eta - 1$ and so $d'_1 + t'_1 \leq 2^\eta - 1$. So, for $D'(\theta)$ that is returned, we have $d'_0, \ldots, d'_{\kappa-2} < 2^\eta$ and $d_{\kappa-1} < 2^\nu$. This shows that $D'(\theta)$ is indeed a proper polynomial.

So, two successive invocations of reduce on $C(\theta)$ reduces it to a proper polynomial. In practice, however, this is not done at each step. Only one invocation is made. As observed above, $\mathsf{reduce}(C(\theta))$ returns $D(\theta)$ for which all coefficients $d_0, d_2, \ldots, d_{\kappa-1}$ satisfy the appropriate bounds and only d_1 can possibly require $\eta + 1$ bits to represent instead of the required η-bit representation. This does not cause any overflow in the intermediate computation and so we do not reduce $D(\theta)$ further. It is only at the end, that an additional invocation of reduce is made to ensure that a proper polynomial is obtained, on which we apply the makeUnique procedure to ensure unique representation of elements of $\mathbb{F}_p$.

4.4 Field Addition

This operation will only be required for the representation using the larger value of κ.

Let $A(\theta) = \sum_{i=0}^{\kappa-1} a_i\theta^i$ and $B(\theta) = \sum_{i=0}^{\kappa-1} b_i\theta^i$ be two polynomials. Let $C(\theta) = \sum_{i=0}^{\kappa-1} c_i\theta^i$ where $c_i = a_i + b_i$ for $i = 0, \ldots, \kappa - 1$. From the bounds on a_i and b_i, we have $c_i < 2^{\eta+1} - 1$ for $i = 0, \ldots, \kappa - 2$ and $c_{\kappa-1} < 2^{\nu+1} - 1$. The operation $\mathsf{sum}(A(\theta), B(\theta))$ is defined to be $D(\theta)$ which is obtained as $D(\theta) = \mathsf{reduce}(C(\theta))$.

4.5 Field Negation

This operation will only be required for the representation using the larger value of κ.

Let $A(\theta) = \sum_{i=0}^{\kappa-1} a_i\theta^i$ be a polynomial. We wish to compute $-A(\theta) \bmod p$. Let $\mathfrak{n}$ be the least integer such that all the coefficients of $2^\mathfrak{n}\mathfrak{P}(\theta) - A(\theta)$ are non-negative. By $\mathsf{negate}(A(\theta))$ we denote $T(\theta) = 2^\mathfrak{n}\mathfrak{P}(\theta) - A(\theta)$. Reducing $T(\theta)$ modulo p gives the desired answer. Let $T(\theta) = \sum_{i=0}^{\kappa-1} t_i\theta^i$ so that $t_i = 2^\mathfrak{n}\mathfrak{p}_i - a_i \geq 0$.

The t_i's are computed using two's complement subtraction. The result of a subtraction can be negative. By ensuring that the t_i's are non-negative, this situation is avoided. Considering all values to be 64-bit quantities, the computation of t_i is done in the following manner: $t_i = ((2^{64} - 1) - a_i) + (1 + 2^\mathfrak{n}\mathfrak{p}_i) \bmod 2^{64}$. The operation $(2^{64} - 1) - a_i$ is equivalent to taking the bitwise complement of a_i, which is equivalent to $1^{64} \oplus a_i$.

From (20), $\mathfrak{p}_0 = 2^\eta - \delta$, $\mathfrak{p}_i = 2^\eta - 1$ for $i = 1, \ldots, \kappa - 2$ and $\mathfrak{p}_{\kappa-1} = 2^\nu - 1$.

1. If $A(\theta)$ is a proper polynomial, then $\mathfrak{n} = 1$ is sufficient to ensure the non-negativity constraint on the coefficients of $T(\theta)$. Using $\mathfrak{n} = 1$, ensures that $t_0, \ldots, t_{\kappa-2} \leq 2\mathfrak{p}_i = 2^{\eta+1} - 2$ and $t_{\kappa-1} \leq 2\mathfrak{p}_{\kappa-1} = 2^{\nu+2} - 2$. So, $t_0, \ldots, t_{\kappa-2}$ can be represented using $\eta + 1$ bits and $t_{\kappa-1}$ can be represented using $\nu + 1$ bits.
2. More generally, suppose that $A(\theta)$ is equal to $2^\mathfrak{r}$ times a proper polynomial. Then choosing $\mathfrak{n} = \mathfrak{r} - \nu + 1$ is sufficient to ensure the non-negativity condition on the coefficients of $T(\theta)$.

Later we explain how the above two situations arise.

Given the operation of negation, subtraction can be done by first negating the subtrahend and then adding to the minuend followed by a reduction.

4.6 Multiplication by a Small Constant

This operation will only be required for the representation using the larger value of κ.

Let $A(\theta) = \sum_{i=0}^{\kappa-1} a_i\theta^i$ be a polynomial and c be a small positive integer considered to be an element of $\mathbb{F}_p$. In our applications, c will be at most 9 bits. The operation $\mathsf{constMult}(A(\theta), c)$ will denote the polynomial $C(\theta) = \sum_{i=0}^{\kappa-1}(ca_i)\theta^i$. We do not apply the algorithm reduce to $C(\theta)$. This is because in our application, multiplication by a constant will be followed by a Hadamard operation and the reduce algorithm is applied after the Hadamard operation. This improves efficiency.

4.7 Field Multiplication

This operation is required for both the larger and the smaller values of κ.

Suppose that $A(\theta) = \sum_{i=0}^{\kappa-1} a_i\theta^i$ and $B(\theta) = \sum_{i=0}^{\kappa-1} b_i\theta^i$ are to be multiplied. Two algorithms for multiplication called mult and multe are defined in Table 8. The reasons for defining two different

input: $\mathsf{mult}(A(\theta), B(\theta))$ **output:** $C(\theta)$ 1. $C(\theta) \leftarrow \mathsf{polyMult}(A(\theta), B(\theta))$; 2. $C(\theta) \leftarrow \mathsf{fold}(C(\theta))$; 3. return $\mathsf{reduce}(C(\theta))$.	**input:** $\mathsf{multe}(A(\theta), B(\theta))$ **output:** $C(\theta)$ 1. $C(\theta) \leftarrow \mathsf{polyMult}(A(\theta), B(\theta))$; 2. $C(\theta) \leftarrow \mathsf{expand}(C(\theta))$; 3. $C(\theta) \leftarrow \mathsf{fold}(C(\theta))$; 4. return $\mathsf{reduce}(C(\theta))$.

Table 8. Field multiplication algorithms.

$\mathsf{expand}(C(\theta))$
input: $C(\theta) = c_0 + c_1\theta + \cdots + c_{2\kappa-2}\theta^{2\kappa-2}$
output: $D(\theta) = d_0 + d_1\theta + \cdots + d_{2\kappa-1}\theta^{2\kappa-1}$
1. for $i = 0, \ldots, \kappa-1$, $d_i \leftarrow c_i$;
2. $s_0 \leftarrow 0$;
3. for $i = 0, \ldots, \kappa-2$, $d_{\kappa+i} \leftarrow \mathsf{lsb}_\eta(c_{\kappa+i} + s_i)$; $s_{i+1} \leftarrow (c_{\kappa+i} + s_i)/2^\eta$;
4. $d_{2\kappa-1} \leftarrow s_{\kappa-1}$;
5. return $D(\theta)$.

Table 9. The expand procedure.

multiplication algorithms are discussed later.

Let $C(\theta)$ be the result of $\mathsf{polyMult}(A(\theta), B(\theta))$. Then $C(\theta)$ can be written as

$$C(\theta) = c_0 + c_1\theta + \cdots + c_{2\kappa-2}\theta^{2\kappa-2} \tag{22}$$

where $c_t = \sum_{s=0}^{t} a_s b_{t-s}$ with the convention that a_i, b_j is zero for $i, j > \kappa - 1$. For $s = 0, \ldots, \kappa-1$, the coefficient $c_{\kappa-1\pm s}$ is the sum of $(\kappa - s)$ products of the form a_ib_j. Since $a_i, b_j < 2^\eta$, it follows that for $s = 0, \ldots, \kappa - 1$,

$$c_{\kappa-1\pm s} \leq (\kappa - s)(2^\eta - 1)^2. \tag{23}$$

Using the representation with the larger value of κ each c_t fits in a 64-bit word and using the representation with the smaller value of κ, each c_t fits in a 128-bit word.

The step polyMult multiplies $A(\theta)$ and $B(\theta)$ as polynomials in θ and returns the result polynomial of degree $2\kappa - 2$. In multe, the step expand is applied to this polynomial and returns a polynomial of degree $2\kappa - 1$. In mult, the step expand is not present and fold is applied to a polynomial of degree $2\kappa - 2$. For uniformity of description, we assume that the input to fold is a polynomial of degree $2\kappa - 1$ where for the case of mult the highest degree coefficient is 0.

The computation of $\mathsf{fold}(C(\theta))$ is the following.

$$\begin{aligned} C(\theta) &= c_0 + c_1\theta + \cdots + c_{\kappa-1}\theta^{\kappa-1} + \theta^{\kappa}\left(c_{\kappa} + c_{\kappa+1}\theta + \cdots + c_{2\kappa-1}\theta^{\kappa-1}\right) \\ &\equiv c_0 + c_1\theta + \cdots + c_{\kappa-1}\theta^{\kappa-1} + 2^{\eta-\nu}\delta\left(c_{\kappa} + c_{\kappa+1}\theta + \cdots + c_{2\kappa-1}\theta^{\kappa-1}\right) \bmod p \\ &= (c_0 + \mathfrak{h}c_{\kappa}) + (c_1 + \mathfrak{h}c_{\kappa+1})\theta + \cdots + (c_{\kappa-1} + \mathfrak{h}c_{2\kappa-1})\theta^{\kappa-1} \end{aligned}$$

where $\mathfrak{h} = 2^{\eta-\nu}\delta$. The polynomial in the last line is the output of $\mathsf{fold}(C(\theta))$.

The expand routine is shown in Table 9. Note that for $D(\theta)$ that is returned by expand we have $d_{\kappa}, \ldots, d_{2\kappa-1} < 2^{\eta}$.

The situations where mult and multe are required are as follows.

1. For $\kappa = 5$, only mult is required.
2. For $p25519$ and $\kappa = 10$, mult will provide an incorrect result. This is because, in this case, some of the coefficients of $\mathsf{fold}(\mathsf{polyMult}(A(\theta), B(\theta)))$ do not fit into 64-bit words. This was already mentioned in [2] and it is for this reason that the "base 2^{26} representation" was discarded. So, for $p25519$ and $\kappa = 10$, only multe will be used.
3. For $p2519$ and $p2663$, both mult and multe will be used at separate places in the scalar multiplication algorithm. This may appear to be strange, since clearly mult is faster than multe. While this is indeed true, the speed improvement is not as much as seems to be apparent from the description of the two algorithms. We mention the following two points.
   - In both mult and multe, as part of fold, multiplication by $\mathfrak{h}$ is required. For the case of mult, the values to which $\mathfrak{h}$ is multiplied are all greater than 32 bits and so the multiplications have to be done using shifts and adds. On the other hand, in the case of multe, the values to which $\mathfrak{h}$ is multiplied are outputs of expand and are hence all less than 32 bits so that these multiplications can be done directly using unsigned integer multiplications. To a certain extent this mitigates the effect of having the expand operation in multe.
   - More importantly, multe is a better choice at one point of the scalar multiplication algorithm. There is a Hadamard operation which is followed by a multiplication. If we do not apply the reduce operation at the end of the Hadamard operation, then the polynomials which are input to the multiplication operation are no longer proper polynomials. Applying mult to these polynomials leads to an overflow after the fold step. Instead, multe is applied, where the expand ensures that there is no overflow at the fold step.

   Due to the combination of the above two effects, the additional cost of the expand operation is more than offset by the savings in eliminating a prior reduce step.

Computation of polyMult: We discuss strategies for polynomial multiplication using the representation for the larger value of κ.

There are several strategies for multiplying two polynomials. For $p2519$, $\kappa = 9$, while for $p25519$ and $p2663$, $\kappa = 10$. Let $C(\theta) = \mathsf{polyMult}(A(\theta), B(\theta))$ where $A(\theta)$ and $B(\theta)$ are proper polynomials. Computing the coefficients of $C(\theta)$ involve 32-bit multiplications and 64-bit additions. The usual measure for assessing the efficacy of a polynomial multiplication algorithm is the number of 32-bit multiplications that would be required. Algorithms from [44] provide the smallest counts of 32-bit multiplication. This measure, however, does not necessarily provide the fastest implementation. Additions and dependencies do play a part and it turns out that an algorithm using a higher number of 32-bit multiplications turn out to be faster in practice. We discuss the cases of $\kappa = 9$ and $\kappa = 10$ separately. In the following, we abbreviate a 32-bit multiplication as [M].

Case $\kappa = 9$: Using 3-3 Karatsuba requires 36[M]. An algorithm given in [44] requires 34[M], but, this algorithm also requires multiplication by small constants which slows down the implementation. We have experimented with several variants and have found the following variant to provide the fastest speed (on the platform for implementation that we used). Consider the 9-limb multiplication to be 8-1 Karatsuba, i.e., the degree 8 polynomial is considered to be a degree 7 polynomial plus the term of degree 8. The two degree 7 (i.e., 8-limb) polynomials are multiplied by 3-level recursive Karatsuba: the 8-limb multiplication is done using 3 4-limb multiplications; each 4-limb multiplication is done using 3 2-limb multiplications; and finally the 2-limb multiplications are done using 4[M] using schoolbook. Using Karatsuba for the 2-limb multiplication is slower. The multiplication by the coefficients of the two degree 8 terms are done directly.

Case $\kappa = 10$: Using binary Karatsuba, this can be broken down into 3 5-limb multiplications. Two strategies for 5-limb multiplications in [44] require 13[M] and 14[M]. The strategy requiring 13[M] also requires multiplications by small constants and turns out to have a slower implementation than the strategy requiring 14[M].

Comparison to previous multiplication algorithm for $p25519$: In the original paper [2] which introduced Curve25519, it was mentioned that for $p25519$, a 10-limb representation using base 2^{26} cannot be used as this leads to an overflow. Instead an approach called "base $2^{25.5}$" was advocated. This approach has been followed in later implementations [7, 13] of Curve25519. In this representation, a 255-bit integer A is written as

$$A = a_0 + 2^{26}a_1 + 2^{51}a_2 + 2^{77}a_3 + 2^{102}a_4 + 2^{128}a_5 + 2^{153}a_6 + 2^{179}a_7 + 2^{204}a_8 + 2^{230}a_9$$

where $a_0, a_2, a_4, a_6, a_8 < 2^{26}$ and $a_1, a_3, a_5, a_7, a_9 < 2^{25}$. Note that this representation cannot be considered as a polynomial in some quantity and so the multiplication of two such representations cannot benefit from the various polynomial multiplication algorithms. Instead, multiplication of two integers A and B in this representation requires all the 100 pairwise multiplications of a_i and b_j along with a few other multiplications by small constants. As mentioned in [13], a total of 109[M] are required to compute the product.

For $p25519$, we have described a 10-limb representation using base as $\theta = 2^{26}$ and have described a multiplication algorithm, namely multe, using this representation. Given the importance of Curve25519, this itself is of some interest. The advantage of multe is that it can benefit from the various polynomial multiplication strategies. On the other hand, the drawback is that the reduction requires a little more time, since the expand step has to be applied.

Following previous work [7], the Sandy2x implementation used SIMD instructions to simultaneously compute two field multiplications. The `vpmuludq` instruction is used to simultaneously carry out two 32-bit multiplications. As a result, the 109 multiplications can be implemented using 54.5 `vpmuludq` instructions per field multiplication.

The multiplication algorithm multe for $p25519$ can also be vectorised using `vpmuludq` to compute two simultaneous field multiplications. We have, however, not implemented this. Since our target is Kummer line computation, we used AVX2 instructions to simultaneously compute four field multiplications. It would be of independent interest to explore the 2-way vectorisation of the new multiplication algorithm for use in the Montgomery curve.

5-limb representation: For $\kappa = 5$, there is not much difference in the multiplication algorithm for $p2519$, $p25519$ and $p2663$. A previous work [6] showed how to perform field arithmetic for $p25519$ using the representation with $\kappa = 5$ and $\eta = \nu = 51$. The Sandy2x code provides an assembly implementation of the multiplication and squaring algorithm and a constant time implementation of the inversion algorithm for $p25519$. The Sandy2x software mentions that the code is basically from [6]. We have used this implementation to perform the inversion required after the Kummer line computation over KL25519$(82, 77)$. We have modified the assembly code for multiplication and squaring over $p25519$ to obtain the respective routines for $p2519$ and $p2663$ which were then used to implement constant time inversion algorithms using fixed addition chains.

4.8 Field Squaring

This operation is required for both the smaller and the larger values of κ.

Let $A(\theta)$ be a proper polynomial. We define $\mathsf{sqr}(A(\theta))$ (resp. $\mathsf{sqre}(A(\theta))$) to be the proper polynomial $C(\theta)$ such that $C(\theta) \equiv A^2(\theta) \bmod p$. The computation of sqr (resp. sqre) is almost the same as that of mult (resp. sqre), except that $\mathsf{polyMult}(A(\theta), B(\theta))$ is replaced by $\mathsf{polySqr}(A(\theta))$ where $\mathsf{polySqr}(A(\theta))$ returns $A^2(\theta)$ as the square of the polynomial $A(\theta)$.

The algorithm sqre is required only for $p25519$ and $\kappa = 10$. In all other cases, the algorithm sqr is required. Unlike the situation for multiplication, there is no situation for either $p2519$ or $p2663$ where sqre is a better option compared to sqr.

4.9 Hadamard Transform

This operation is required only for the representation using the larger value of κ.

Let $A_0(\theta)$ and $A_1(\theta)$ be two polynomials. By $\mathcal{H}(A_0(\theta), A_1(\theta))$ we denote the pair $(B_0(\theta), B_1(\theta))$ where

$$\begin{aligned} B_0(\theta) &= \mathsf{reduce}(A_0(\theta) + A_1(\theta)); \\ B_1(\theta) &= \mathsf{reduce}(A_0(\theta) - A_1(\theta)) = \mathsf{reduce}(A_0(\theta) + \mathsf{negate}(A_1(\theta))). \end{aligned}$$

In our context, there is an application of the Hadamard transform to the output of multiplication by constant. Since the output of multiplication by constant is not reduced, the coefficients of the input polynomials to the Hadamard transform do not necessarily respect the bounds required for proper polynomials. As explained earlier, the procedure negate works correctly even with looser bounds on the coefficients of the input polynomial.

We define the operation unreduced-$\mathcal{H}(A_0(\theta), A_1(\theta))$ which is the same as $\mathcal{H}(A_0(\theta), A_1(\theta))$ except that the reduce operations are dropped. So, the outputs of unreduced-$\mathcal{H}(A_0(\theta), A_1(\theta))$ are not necessarily proper polynomials. If the inputs are proper polynomials, then it is not difficult to see that the first $\kappa - 1$ coefficients of the two output polynomials are at most $\eta + 1$ bits and the last coefficients are at most $\nu + 1$ bits. Leaving the output of the Hadamard operation unreduced saves time. In the scalar multiplication algorithm, in one case this can be done and is followed by the multe operation which ensures that there is no eventual overflow.

4.10 Field Inversion

This operation is required only for the representation using the smaller value of κ.

Suppose the inversion of $A(\theta)$ is required. Inversion is computed in constant time using a fixed addition chain to compute $A(\theta)^{p-2} \bmod p$. This computation boils down to computing a fixed number of squarings and multiplications.

In our context, field inversion is required only for conversion from projective to affine coordinates. The output of the scalar multiplication is in projective coordinates and if for some application the output is required in affine coordinates, then only a field inversion is required. The timing measurements that we report later includes the time required for inversion.

As mentioned earlier, the entire Kummer line scalar multiplication is done using the larger value of κ. Before performing the inversion, the operands are converted to the representation using the smaller value of κ. For $p25519$, the actual inversion is done using the constant time code for inversion used for Curve25519 in the Sandy2x implementation while for $p2519$ and $p2663$, appropriate modifications of this code are used.

5 Vector Operations

While considering vector operations, we consider the representation of field elements using the larger value of κ.

SIMD instructions in modern processors allow parallelism where the same instruction can be applied to multiple data. To take advantage of SIMD instructions it is convenient to organise the data as vectors. The Intel instructions that we target apply to 256-bit registers which are considered to be 4 64-bit words (or, as 8 32-bit words). So, we consider vectors of length 4.

Let $\mathbf{A}(\theta) = (A_0(\theta), A_1(\theta), A_2(\theta), A_3(\theta))$ where $A_k(\theta) = \sum_{i=0}^{\kappa-1} a_{k,i}\theta^i$ are proper polynomials. We will say that such an $\mathbf{A}(\theta)$ is a proper vector. So, $\mathbf{A}(\theta)$ is a vector of 4 elements of $\mathbb{F}_p$. Recall that each $a_{k,i}$ is stored in a 64-bit word. Conceptually one may think of $\mathbf{A}(\theta)$ to be given by a $\kappa \times 4$ matrix of 64-bit words.

We describe a different way to consider $\mathbf{A}(\theta)$. Let $\mathbf{a}_i = (a_{0,i}, a_{1,i}, a_{2,i}, a_{3,i})$ and define $\mathbf{a}_i\theta^i = (a_{0,i}\theta^i, a_{1,i}\theta^i, a_{2,i}\theta^i, a_{3,i}\theta^i)$. Then we can write $\mathbf{A}(\theta)$ as $\mathbf{A}(\theta) = \sum_{i=0}^{\kappa-1} \mathbf{a}_i\theta^i$. Each $\mathbf{a}_i$ is stored as a 256-bit value. We define the following operations.

- $\mathsf{pack}(a_0, a_1, a_2, a_3)$: returns a 256-bit quantity $\mathbf{a}$. Here each a_i is a 64-bit quantity and $\mathbf{a}$ is obtained by concatenating a_0, a_1, a_2, a_3.
- $\mathsf{unpack}(\mathbf{a})$: returns (a_0, a_1, a_2, a_3). Here $\mathbf{a}$ is a 256-bit quantity and the a_i's are 64-bit quantities such that $\mathbf{a}$ is the concatenation of a_0, a_1, a_2, a_3.
- $\mathsf{pack}(A_0(\theta), A_1(\theta), A_2(\theta), A_3(\theta))$: returns $\mathbf{A}(\theta)$ represented as $\mathbf{A}(\theta) = \sum_{i=0}^{\kappa-1} \mathbf{a}_i\theta^i$, where $\mathbf{a}_i = \mathsf{pack}(a_{i,0}, a_{i,1}, a_{i,2}, a_{i,3})$.
- $\mathsf{unpack}(\mathbf{A}(\theta))$: returns $(A_0(\theta), A_1(\theta), A_2(\theta), A_3(\theta))$, where $\mathbf{A}(\theta) = \sum_{i=0}^{\kappa-1} \mathbf{a}_i\theta^i$, for $j = 0, 1, 2, 3$, $A_j(\theta) = \sum_{i=0}^{\kappa-1} a_{j,i}\theta^i$ and $(a_{0,i}, a_{1,i}, a_{2,i}, a_{3,i}) = \mathsf{unpack}(\mathbf{a}_i)$.

In the above, we use pack to denote both the packing of 4 64-bit words into a 256-bit quantity and also the limb-wise packing of four field elements into a vector. Similar overloading of notation is used for unpack.

We define the following vector operations. The operand $\mathbf{A}(\theta)$ represents $(A_0(\theta), A_1(\theta), A_2(\theta), A_3(\theta))$ and similarly, $\mathbf{B}(\theta)$ represents $(B_0(\theta), B_1(\theta), B_2(\theta), B_3(\theta))$.

- $\mathsf{reduce}(\mathbf{A}(\theta))$: returns $(\mathsf{reduce}(A_0(\theta)), \mathsf{reduce}(A_1(\theta)), \mathsf{reduce}(A_2(\theta)), \mathsf{reduce}(A_3(\theta)))$.
- $\mathcal{M}^4(\mathbf{A}(\theta), \mathbf{B}(\theta))$: returns $\mathbf{C}(\theta) = \sum_{i=0}^{\kappa-1} \mathbf{c}_i\theta^i$ representing $(C_0(\theta), C_1(\theta), C_2(\theta), C_3(\theta))$ where $C_k(\theta) = \mathsf{mult}(A_k(\theta), B_k(\theta))$ for $k = 0, \ldots, 3$.
- $\mathcal{ME}^4(\mathbf{A}(\theta), \mathbf{B}(\theta))$: returns $\mathbf{C}(\theta) = \sum_{i=0}^{\kappa-1} \mathbf{c}_i\theta^i$ representing $(C_0(\theta), C_1(\theta), C_2(\theta), C_3(\theta))$ where $C_k(\theta) = \mathsf{multe}(A_k(\theta), B_k(\theta))$ for $k = 0, \ldots, 3$.
- $\mathcal{S}^4(\mathbf{A}(\theta))$: returns $\mathbf{C}(\theta) = \sum_{i=0}^{\kappa-1} \mathbf{c}_i\theta^i$ representing $(C_0(\theta), C_1(\theta), C_2(\theta), C_3(\theta))$ where $C_k(\theta) = \mathsf{sqr}(A_k(\theta))$ for $k = 0, \ldots, 3$.
- $\mathcal{SE}^4(\mathbf{A}(\theta))$: returns $\mathbf{C}(\theta) = \sum_{i=0}^{\kappa-1} \mathbf{c}_i\theta^i$ representing $(C_0(\theta), C_1(\theta), C_2(\theta), C_3(\theta))$ where $C_k(\theta) = \mathsf{sqre}(A_k(\theta))$ for $k = 0, \ldots, 3$.
- $\mathcal{C}^4(\mathbf{A}(\theta), \mathbf{d})$: returns $\mathbf{C}(\theta) = \sum_{i=0}^{\kappa-1} \mathbf{c}_i\theta^i$ representing $(C_0(\theta), C_1(\theta), C_2(\theta), C_3(\theta))$ where $\mathbf{d} = (d_0, d_1, d_2, d_3)$; $C_k(\theta) = \mathsf{constMult}(A_k(\theta), d_k)$ for $k = 0, \ldots, 3$. Recall that the output of constMult is not reduced and so neither is the output of $\mathcal{C}^4$.

The operation $\mathcal{ME}^4$ differs from $\mathcal{M}^4$ in the use of multe instead of mult to perform the multiplications. Similarly, $\mathcal{SE}^4$ differs from $\mathcal{S}^4$ in the use of sqre instead of sqr to perform squarings. The key Intel intrinsics operations that are required to implement the above vector operations are the following.

- `_mm256_add_epi64`: On inputs $\mathbf{a} = (a_0, a_1, a_2, a_3)$ and $\mathbf{b} = (b_0, b_1, b_2, b_3)$, returns $(a_0+b_0, a_1+b_1, a_2+b_2, a_3+b_3)$ with each component reduced modulo 2^{64}.
- `_mm256_sub_epi64`: On inputs $\mathbf{a} = (a_0, a_1, a_2, a_3)$ and $\mathbf{b} = (b_0, b_1, b_2, b_3)$, returns $(a_0-b_0, a_1-b_1, a_2-b_2, a_3-b_3)$ with each component reduced modulo 2^{64}. We have used this operation only in context of Karatsuba multiplication, i.e., for a subtraction of the type $(a+b)(c+d)-(ac+bd) = ad+bc$ for non-negative integers a, b, c and d. The result is guaranteed to be non-negative and so there is no need to handle the sign.
- `_mm256_mul_epu32`: On inputs $\mathbf{a} = (a_0, a_1, a_2, a_3)$ and $\mathbf{b} = (b_0, b_1, b_2, b_3)$, returns $(a_0b_0, a_1b_1, a_2b_2, a_3b_3)$ with each component reduced modulo 2^{64}.

5.1 Vector Hadamard Operation

The Hadamard operation $\mathcal{H}(A(\theta), B(\theta))$ is required to output $(C(\theta), D(\theta))$ where $C(\theta) \equiv A(\theta) + B(\theta) \bmod p$ and $D(\theta) \equiv A(\theta) - B(\theta) \bmod p$. We define the vector extension of the Hadamard operation, which computes two simultaneous Hadamard operations using SIMD vector instructions. For a 256-bit quantity $\mathbf{a} = (a_0, a_1, a_2, a_3)$ we define $\mathsf{dup}_1(\mathbf{a}) = (a_0, a_0, a_2, a_2)$ and $\mathsf{dup}_2(\mathbf{a}) = (a_1, a_1, a_3, a_3)$.

The vector Hadamard operation $\mathcal{H}^2$ is shown in Table 10. The Hadamard operation involves a subtraction. As explained in Section 4.5 this is handled by first computing a negation followed by an addition. Negation of a polynomial is computed as subtracting the given polynomial from $2^{\mathfrak{n}}\mathfrak{P}(\theta)$ where $\mathfrak{n}$ is chosen to ensure that all the coefficients of the result are positive.

$\mathcal{H}^2(\mathbf{A}(\theta))$
input: $\mathbf{A}(\theta) = \sum_{i=0}^{\kappa-1} \mathbf{a}_i\theta^i$ representing $(A_0(\theta), A_1(\theta), A_2(\theta), A_3(\theta))$;
output: $\mathbf{C}(\theta) = \sum_{i=0}^{\kappa-1} \mathbf{c}_i\theta^i$ representing $(A_0(\theta)+A_1(\theta), A_0(\theta)-A_1(\theta), A_2(\theta)+A_3(\theta), A_2(\theta)-A_3(\theta))$
with each component reduced modulo p;
1. for $i = 0, \ldots, \kappa-1$ do
2. $\mathbf{s} = \mathsf{dup}_1(\mathbf{a}_i)$;
3. $\mathbf{t} = \mathsf{dup}_2(\mathbf{a}_i)$;
4. $\mathbf{t} = \mathbf{t} \oplus (0^{64}, 1^{64}, 0^{64}, 1^{64})$;
5. $\mathbf{t} = \mathbf{t} + (0^{64}, 2^{\mathfrak{n}}\mathfrak{p}_i + 1, 0^{64}, 2^{\mathfrak{n}}\mathfrak{p}_i + 1)$;
6. $\mathbf{c}_i = \mathbf{t} + \mathbf{s}$;
7. end for;

return $\mathsf{reduce}(\mathbf{C}(\theta))$.

Table 10. Vector Hadamard operation.

The operations dup_1 and dup_2 are implemented using `_mm256_permute4x64_epi64`; $\oplus$ is implemented using `_mm256_xor_si256`; the additions in Steps 5 and 6 are implemented using `_mm256_add_epi32`;

1. The operation $\mathcal{C}^4$ (which is the vector version of $\mathsf{constMult}$) multiplies the input proper polynomials with constant and the result is not reduced (since the output of $\mathsf{constMult}$ is not reduced). The constant is one of the parameters A^2 and B^2 of the Kummer line. The output of $\mathcal{C}^4$ forms the input to $\mathcal{H}^2$. Choosing $\mathfrak{n} \geq \lceil \log_2 \max(A^2, B^2) \rceil$ ensures the non-negativity condition for the subtraction operation.
2. We define a unreduced version of $\mathcal{H}^2$ to be $\mathsf{unreduced}\text{-}\mathcal{H}^2$. This procedure is the same as $\mathcal{H}^2$ except that at the end instead of returning $\mathsf{reduce}(\mathbf{C}(\theta))$, $\mathbf{C}(\theta)$ is returned. Following the discussion in

Section 4.5, to apply the procedure unreduced-$\mathcal{H}^2$ to a proper polynomial it is sufficient to choose $\mathfrak{n} = 1$. The first $\kappa - 2$ coefficients of the output can be represented using $\eta + 1$ bits and the last coefficient can be represented using $\nu + 1$ bits.

5.2 Vector Duplication

Let $\mathbf{a} = (a_0, a_1, a_2, a_3)$ and $\mathfrak{b}$ be a bit. We define an operation $\mathsf{copy}(\mathbf{a}, \mathfrak{b})$ as follows: if $\mathfrak{b} = 0$, return (a_0, a_1, a_0, a_1); and if $\mathfrak{b} = 1$, return (a_2, a_3, a_2, a_3). The operation copy is implemented using the instruction `_mm256_permutevar8x32_epi32`.

Let $\mathbf{A}(\theta) = \sum_{i=0}^{\kappa-1} \mathbf{a}_i \theta^i$ be a proper vector and b be a bit. We define the operation $\mathcal{P}^4(\mathbf{A}, \mathfrak{b})$ to return $\sum_{i=0}^{\kappa-1} \mathsf{copy}(\mathbf{a}_i, b)\theta^i$. If $\mathbf{A}(\theta)$ represents $(A_0(\theta), A_1(\theta), A_2(\theta), A_3(\theta))$, then

$$\mathcal{P}^4(\mathbf{A}, \mathfrak{b}) = \begin{cases} (A_0(\theta), A_1(\theta), A_0(\theta), A_1(\theta)) \text{ if } \mathfrak{b} = 0; \\ (A_2(\theta), A_3(\theta), A_2(\theta), A_3(\theta)) \text{ if } \mathfrak{b} = 1. \end{cases}$$

6 Vectorised Scalar Multiplication

Scalar multiplication on the Kummer line is computed from a base point represented as $[x^2 : z^2]$ in the square only setting and an ℓ-bit non-negative integer n. The quantities x^2 and z^2 are elements of $\mathbb{F}_p$ and we write their representations as $X(\theta)$ and $Z(\theta)$. If x^2 and z^2 are small as in the fixed base points of the Kummer lines, then $X(\theta)$ and $Z(\theta)$ have 1-limb representations. In general, the field elements $X(\theta)$ and $Z(\theta)$ will be arbitrary elements of $\mathbb{F}_p$ and will have a 9-limb (for $p2519$) or a 10-limb (for $p25519$ and $p2663$) representation.

Vectorised scalar multiplications for the primes $p2519$ and $p2663$ are shown in Table 11. Algorithm $\mathsf{scalarMult}(P, n)$ is the variable base scalar multiplication algorithm and can be applied for all P, whereas, Algorithm $\mathsf{scalarMultFB}(P, n)$ is the fixed base scalar multiplication algorithm and can be applied when the base point $P = [X(\theta) : Z(\theta)]$ is fixed and small. Modifications required for $p25519$ are mentioned later. The differences between $\mathsf{scalarMult}(P, n)$ and $\mathsf{scalarMultFB}(P, n)$ are the following.

1. In Step 7, $\mathsf{scalarMult}(P, n)$ uses the operation unreduced-$\mathcal{H}^2$, whereas, $\mathsf{scalarMultFB}(P, n)$ uses the operation $\mathcal{H}^2$.
2. In Step 13, $\mathsf{scalarMult}(P, n)$ uses the operation $\mathcal{M}^4$, whereas, $\mathsf{scalarMultFB}(P, n)$ uses the operation unreduced-$\mathcal{C}^4$.

For $p2663$, in Step 9 of $\mathsf{scalarMultFB}(P, n)$ it is possible to use $\mathcal{M}^4$ instead of $\mathcal{ME}^4$.

An inversion is required at Step 15. The representations of $U(\theta)$ and $V(\theta)$ are first converted to the one using the smaller value of κ. Let these be denoted as u and v. The computation of u/v is as follows: first $w = v^{-1}$ is computed and then $x = w \cdot u$ are computed. As mentioned in Section 4.10, the inversion is computed in constant time. The multiplications and squarings in this computation are performed using the representation with $\kappa = 5$ so that both w and x are also represented using $\kappa = 5$. A final reduce call is made on x followed by a makeUnique call whose output is returned.

Modifications for $p25519$: All occurrences of $\mathcal{M}^4$ and $\mathcal{S}^4$ in scalarMult and scalarMultFB are replaced by $\mathcal{ME}^4$ and $\mathcal{SE}^4$ respectively.

Correctness: Steps 7 to 13 constitute a single vectorised ladder step. At the i-th iteration, suppose $(kP, (k+1)P)$, for some k, is a pair of points which forms the input to the ladder step. If $n_i = 0$, then the output of the ladder step is the pair of points $(2kP, (2k+1)P)$; and if $n_i = 1$, then the output of the ladder step is the pair of points $((2k+1)P, (2k+2)P)$. Suppose $kP = [X_1(\theta) : Z_1(\theta)]$ and $(k+1)P = [X_2(\theta) :$

$\mathsf{scalarMult}(P, n)$	$\mathsf{scalarMultFB}(P, n)$
Input: base point $P = [X(\theta) : Z(\theta)]$;	Input: base point $P = [X(\theta) : Z(\theta)]$;
$\mathfrak{l}$-bit scalar n given as $(1, n_{\mathfrak{l}-2}, \ldots, n_0)$;	$\mathfrak{l}$-bit scalar n given as $(1, n_{\mathfrak{l}-2}, \ldots, n_0)$;
Output: $U(\theta)/V(\theta)$ where $nP = [U(\theta) : V(\theta)]$;	Output: $U(\theta)/V(\theta)$ where $nP = [U(\theta) : V(\theta)]$;
1. $\mathfrak{a} = \mathsf{pack}(B^2, A^2, B^2, A^2)$;	1. $\mathfrak{a} = \mathsf{pack}(B^2, A^2, B^2, A^2)$;
2. $\mathfrak{c}_0 = \mathsf{pack}(b^2, a^2, Z, X)$;	2. $\mathfrak{c}_0 = \mathsf{pack}(b^2, a^2, Z, X)$;
3. $\mathfrak{c}_1 = \mathsf{pack}(Z, X, b^2, a^2)$;	3. $\mathfrak{c}_1 = \mathsf{pack}(Z, X, b^2, a^2)$;
4. compute $2P = (X_2(\theta), Z_2(\theta))$;	4. compute $2P = (X_2(\theta), Z_2(\theta))$;
5. $\mathbf{T}(\theta) = \mathsf{pack}(X(\theta), Z(\theta), X_2(\theta), Z_2(\theta))$;	5. $\mathbf{T}(\theta) = \mathsf{pack}(X(\theta), Z(\theta), X_2(\theta), Z_2(\theta))$;
6. for $i = \mathfrak{l} - 2$ down to 0	6. for $i = \mathfrak{l} - 2$ down to 0
7. $\mathbf{T}(\theta) = \mathsf{unreduced}\text{-}\mathcal{H}^2(\mathbf{T}(\theta))$;	7. $\mathbf{T}(\theta) = \mathcal{H}^2(\mathbf{T}(\theta))$;
8. $\mathbf{S}(\theta) = \mathcal{P}^4(\mathbf{T}(\theta), n_i)$;	8. $\mathbf{S}(\theta) = \mathcal{P}^4(\mathbf{T}(\theta), n_i)$;
9. $\mathbf{T}(\theta) = \mathcal{ME}^4(\mathbf{T}(\theta), \mathbf{S}(\theta))$;	9. $\mathbf{T}(\theta) = \mathcal{ME}^4(\mathbf{T}(\theta), \mathbf{S}(\theta))$;
10. $\mathbf{T}(\theta) = \mathcal{C}^4(\mathbf{T}(\theta), \mathfrak{a})$;	10. $\mathbf{T}(\theta) = \mathcal{C}^4(\mathbf{T}(\theta), \mathfrak{a})$;
11. $\mathbf{T}(\theta) = \mathcal{H}^2(\mathbf{T}(\theta))$;	11. $\mathbf{T}(\theta) = \mathcal{H}^2(\mathbf{T}(\theta))$;
12. $\mathbf{T}(\theta) = \mathcal{S}^4(\mathbf{T}(\theta))$;	12. $\mathbf{T}(\theta) = \mathcal{S}^4(\mathbf{T}(\theta))$;
13. $\mathbf{T}(\theta) = \mathcal{M}^4(\mathbf{T}(\theta), \mathfrak{c}_{n_i})$;	13. $\mathbf{T}(\theta) = \mathcal{C}^4(\mathbf{T}(\theta), \mathfrak{c}_{n_i})$;
14. end for;	14. end for;
15. $(U(\theta), V(\theta), \cdot, \cdot) = \mathsf{unpack}(\mathsf{reduce}(\mathbf{T}(\theta)))$;	15. $(U(\theta), V(\theta), \cdot, \cdot) = \mathsf{unpack}(\mathsf{reduce}(\mathbf{T}(\theta)))$;
16. return $U(\theta)/V(\theta)$.	16. return $U(\theta)/V(\theta)$.

Table 11. Variable and fixed base vectorised scalar multiplication algorithm for $p2519$ and $p2663$. For scalarMult, the base point $[X(\theta) : Z(\theta)]$ is not necessarily small whereas, for scalarMultFB, the base point $[X(\theta) : Z(\theta)]$ is fixed and small. Recall that $A^2 = a^2 + b^2$ and $B^2 = a^2 - b^2$.

$Z_2(\theta)]$. These two points are represented in packed form as $(X_1(\theta), Z_1(\theta), X_2(\theta), Z_2(\theta))$ which is the vector input to the ladder step. Denoting the output of the ladder step as $(X_3(\theta), Z_3(\theta), X_4(\theta), Z_4(\theta))$, the operation of the ladder step is shown in Figure 4. The correctness of the ladder step is easy to argue from which the correctness of the vectorised scalar multiplication follows.

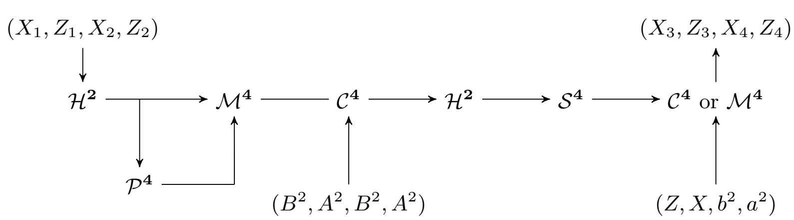

Fig. 4. One vectorised ladder step for fixed base scalar multiplication.

7 Implementation and Timings

We have implemented the vectorised scalar multiplication algorithm in 64-bit AVX2 intrinsics instructions. The code implements the vectorised ladder algorithm which takes the same amount of time for all scalars. Consequently, our code also runs in constant time. The code is publicly available at [25].

Timing experiments were carried out on a single core of the following two platforms.

Haswell: Intel®Core™i7-4790 4-core CPU @ 3.60GHz.
Skylake: Intel®Core™i7-6700 4-core CPU @ 3.40GHz.

In both cases, the OS was 64-bit Ubuntu-16.04 LTS and the C code was compiled using GCC version 5.4.0. During timing measurements, turbo boost and hyperthreading were turned off. An initial cache warming was done with 25000 iterations and then the median of 100000 iterations was recorded. The Time Stamp Counter (TSC) was read from the CPU to RAX and RDX registers by RDTSC instruction.

Table 12 compares the number of cycles required by our implementation with that of a few other concrete curve proposals. All the timings are for constant time code on the Haswell processor using variable base scalar multiplication. For Four-$\mathbb{Q}$, $\mathcal{K}_{11,-22,-19,-3}$ and the results from [49] and [33], the timings are obtained from the respective papers. For Curve25519, we downloaded the `Sandy2x`[7] library and measured the performance using the methodology from [32]. The cycle count of 156060 that we obtain for variable base scalar multiplication for Curve25519 on Haswell approximately matches the 156076 cycles reported by Tung Chou[8] and the count of about 156500 cycles reported in [22]. Further, EBACS[9] reports approximately 156000 cycles on the machine `titan0`.

curve	genus	security	field	endomorphism	cycles	pre-comp tab
Curve25519 [13]	1	126	$\mathbb{F}_{2^{255}-19}$	no	156060	no
NIST P-256 [33]	1	128	$\mathbb{F}_{2^{256}-2^{224}+2^{192}+2^{96}-1}$	no	291000	no
Four-$\mathbb{Q}$ [16][10]	1	123	$\mathbb{F}_{(2^{127}-1)^2}$	yes	59000	2048 bits
				no	109000	no
$\mathcal{K}_{11,-22,-19,-3}$ [4][11]	2	125	$\mathbb{F}_{2^{127}-1}$	no	60468	no
Koblitz [49]	1	128	$\mathbb{F}_{4^{149}}$	yes	69656	4768 bits
KL2519(81, 20)	1	124	$\mathbb{F}_{2^{251}-9}$	no	110408	no
KL25519(82, 77)	1	125.7	$\mathbb{F}_{2^{255}-19}$	no	146656	no
KL2663(260, 139)	1	131.2	$\mathbb{F}_{2^{266}-3}$	no	155460	no

Table 12. Timing comparison for variable base scalar multiplication on Haswell. The entries are cycle counts. The references point to the best known implementations. Curve25519 was proposed in [2]; NIST P-256 was proposed in [46]; the curve used in [49] was proposed in [41]; and $\mathcal{K}_{11,-22,-19,-3}$ was proposed in [31].

curve	security	Haswell		Skylake	
		fixed base	var base	fixed base	var base
Curve25519 [13]	126	144224	156060	126498	136774
KL2519(81,20)	124	93020	110408	78628	92976
KL25519(82,77)	125.7	116832	146656	93578	123102
KL2663(260,139)	131.2	119564	155460	91130	121958

Table 13. Timing comparison of Kummer lines with Curve25519 on Haswell and Skylake platforms. The entries are cycle counts.

Timing results on Haswell and Skylake platforms for Curve25519 and the Kummer lines for both fixed base and variable base scalar multiplications are shown in Table 13.

Fixed base scalar multiplication can achieve efficiency improvements in two possible ways. One, by using a base point with small coordinates and two, by using pre-computation. We have used only

[7] Downloaded from `https://bench.cr.yp.to/supercop/supercop-20160910.tar.xz` (last accessed on September 1, 2018). We used `crypto_scalarmult(q,n,p)` to measure variable base scalar multiplication and `crypto_scalarmult_base(q,n)` to measure fixed base scalar multiplication.

[8] `https://moderncrypto.org/mail-archive/curves/2015/000637.html`, accessed on September 1, 2018.

[9] `https://bench.cr.yp.to/results-dh.html`, accessed on September 1, 2018.

[10] Improved timing results of 54000 and 104000 respectively for implementation with and without endomorphism for Four-$\mathbb{Q}$ have been reported in the extended version `http://eprint.iacr.org/2015/565.pdf` (accessed on September 1, 2018).

[11] The original speed reported in [4] was 54389. The figure 60468 is reported to be the median cycles per byte at `https://bench.cr.yp.to/results-dh.html` (accessed on September 1, 2018)) for the machine `titan0`. We refer to `http://eprint.iacr.org/2015/565.pdf` for a possible explanation of the discrepancy.

the first method. Using pre-computed tables, [33] reports much faster timing for NIST P-256 and [13] reports much faster timing for Curve25519. We have not investigated the use of pre-computed tables to speed up fixed base scalar multiplication for Kummer lines.

Based on entries in Table 13, we conclude the following. We use the shorthands $K_1 := \mathsf{KL2519}(81, 20)$, $K_2 := \mathsf{KL25519}(82, 77)$ and $K_3 := \mathsf{KL2663}(260, 139)$.

1. K_1 and K_2 are faster than Curve25519 on both Haswell and Skylake processors for both fixed base and variable base scalar multiplications. In particular, we note that even though Curve25519 and K_2 use the same underlying prime $p25519$, K_2 provides speed improvements over Curve25519. This points to the fact that the Kummer line is more SIMD friendly than the Montgomery curve.
2. On the Skylake processor, K_3 is faster than Curve25519 for both fixed base and variable base scalar multiplications. On the earlier Haswell processor, K_3 is faster than Curve25519 for fixed base scalar multiplication while both K_3 and Curve25519 take roughly the same time for variable base scalar multiplication. We note that speed improvements for fixed base scalar multiplication does not necessarily imply speed improvement for variable base scalar multiplication, since the code optimisations in the two cases are different.
3. In terms of security, K_3 offers the highest security followed by Curve25519, K_2 and K_1 in that order. The security gap between K_3 and Curve25519 is 5.2 bits; between Curve25519 and K_2 is 0.3 bits; and between Curve25519 and K_1 is 2 bits.

Multiplication and squaring using the 5-limb representation take roughly the same time for all the three primes $p2519$, $p25519$ and $p2663$. So, the comparative times for inversion modulo these three primes is determined by the comparative sizes of the corresponding addition chains. As a result, the time for inversion is the maximum for $p2663$, followed by $p25519$ and $p2519$ in that order.

Curve25519 is based upon $p25519$ and so the inversion step for Curve25519 is faster than that for K_3. Further, the scalars for K_3 are about 8 bits longer than those for Curve25519. It is noticeable that despite these two facts, other than variable base scalar multiplication on Haswell, a scalar multiplication over K_3 is faster than that over Curve25519. This is due to the structure of the primes $p2663 = 2^{266} - 3$ and $p25519 = 2^{255} - 19$ where 3 being smaller than 19 allows significantly faster multiplication and squaring in the 10-limb representations of these two primes.

On the Skylake processor, K_3 provides both higher speed and higher security compared to Curve25519 If one is interested in obtaining the maximum security, then K_3 should be used. On the other hand, if one considers 124 bits of security to be adequate, then K_1 should be used. The only reason for considering the prime $p25519$ in comparison to either $p2519$ or $p2663$ is that 255 is closer to a multiple of 32 than either of 251 or 266. If public keys are transmitted as 32-bit words, then the wastage of bits would be minimum for $p25519$ compared to $p2519$ or $p2663$. Whether this is an overriding reason for discarding the higher security and higher speed offered by $p2663$ or the much higher speed and small loss in security offered by $p2519$ would probably depend on the application at hand. If for some reason, $p25519$ is preferred to be used, then K_2 offers higher speed than Curve25519 at a loss of only 0.3 bits of security.

We have comprehensively considered the different possibilities for algorithmic improvements to the basic idea leading to significant reductions in operations count. At this point of time, we do not see any way of further reducing the operation counts. On the other hand, we note that our implementations of the Kummer line scalar multiplications are based on Intel intrinsics. There is a possibility that a careful assembly implementation will further improve the speed.

8 Implementation of qDSA on Kummer Lines

A key step in the verification algorithm of qDSA is a predicate called Check. To use qDSA for Kummer lines, we first define the Check predicate for a Legendre form curve $E_\mu : y^2 = x(x-1)(x-\mu)$. Suppose

$\mathbf{P} = [X_P : \cdot : Z_P]$, $\mathbf{Q} = [X_Q : \cdot : Z_Q]$ and $\mathbf{R} = [X_R : \cdot : Z_R]$ are three points (in projective coordinates) on E_μ. The y-coordinates of these points are unknown. The Check predicate returns true if and only if $[X_R : \cdot : Z_R]$ is equal to either $[X_S : \cdot : Z_S]$ or $[X_D : \cdot : Z_D]$ where $[X_S : Y_S : Z_S] = \mathbf{P} + \mathbf{Q}$ and $[X_D : Y_D : Z_D] = \mathbf{P} - \mathbf{Q}$ for some $Y_S, Y_D \in \mathbb{F}_p$. Following the derivation of Proposition 3 of [50], it is possible to derive the Check operation on Legendre form curves. (In the code package [26] corresponding to qDSA implementations on the Kummer line, we provide the SAGE file which shows the necessary derivations.) Define $\mathfrak{C}_{L,\mu}(\mathbf{P}, \mathbf{Q}, \mathbf{R})$ as follows.

$$\mathfrak{C}_{L,\mu}(\mathbf{P}, \mathbf{Q}, \mathbf{R}) = \alpha_0 X_R^2 - 2\alpha_1 X_R Z_R + \alpha_2 Z_R^2, \tag{24}$$

where

$$\begin{aligned} \alpha_0 &= (X_P Z_Q - Z_P X_Q)^2; \\ \alpha_1 &= (X_P Z_Q + X_Q Z_P)(X_P X_Q + \mu Z_P Z_Q) - 2(\mu + 1) X_P X_Q Z_P Z_Q; \\ \alpha_2 &= (X_P X_Q - \mu Z_P Z_Q)^2. \end{aligned} \tag{25}$$

The predicate $\mathsf{Check}_{L,\mu}(\mathbf{P}, \mathbf{Q}, \mathbf{R})$ on Legendre form curve returns T if and only if $\mathfrak{C}_{L,\mu}(\mathbf{P}, \mathbf{Q}, \mathbf{R}) = 0$.

Our goal is to instantiate qDSA on Kummer lines. So, we require the Check operation to be defined on a Kummer line. Let $\mathcal{K}_{a^2,b^2}$ be a Kummer line associated with a Legendre form curve E_μ, where $\mu = a^4/(a^4 - b^4)$. Given a point $P = [x^2 : z^2]$ on $\mathcal{K}_{a^2,b^2}$, recall that the map $\psi(P)$ given in (15) maps to a point $\mathbf{P}$ in E_μ. The basic idea of defining the Check operation on a Kummer line is to map the points P, Q and R into the Legendre form curve using ψ and then apply the Check operation on the Legendre form curve. One issue is that the map ψ determines a point E_μ only up to addition by $\mathbf{T} = [\mu : 0 : 1]$. So, the predicate $\mathsf{Check}_{K,a^2,b^2}$ on $\mathcal{K}_{a^2,b^2}$ is defined as follows. Given points $P = [x_P^2 : z_P^2]$, $Q = [x_Q^2 : z_Q^2]$ and $R = [x_R^2 : z_R^2]$ on $\mathcal{K}_{a^2,b^2}$, $\mathsf{Check}_{K,a^2,b^2}(P, Q, R)$ returns T if and only if

$$\text{either } \mathsf{Check}_{L,\mu}(\psi(P), \psi(Q), \psi(R)) \text{ returns } \mathsf{T}, \text{ or, } \mathsf{Check}_{L,\mu}(\psi(P), \psi(Q), \psi(R) + \mathbf{T}) \text{ returns } \mathsf{T}. \tag{26}$$

The parameters of qDSA on a Kummer line are as follows.

1. A Kummer line $\mathcal{K}_{a^2,b^2}$.
2. A base point $P = [x_P^2 : 1]$ of order ℓ. The points P and the corresponding values of $\lg \ell$ for the Kummer lines considered in this work are shown in Table 4.
3. The point $\mathbf{T} = [\mu : 1 : 0]$ is a point of order two on the Legendre form curve E_μ corresponding to the Kummer line $\mathcal{K}_{a^2,b^2}$.
4. A hash function H which produces a 64-byte digest. We have used SHAKE128 [20] to instantiate H.
5. An integer $\mathfrak{b}$, where $\mathfrak{b} = 32$ for $\mathsf{KL2519}(81, 20)$ and $\mathsf{KL25519}(82, 77)$ and $\mathfrak{b} = 33$ for $\mathsf{KL2663}(260, 139)$.
6. The private key is a pair (d', d'') where d' is a $\mathfrak{b}$-byte quantity and d'' is a $(64 - \mathfrak{b})$-byte quantity. The public key is x_Q^2 which is the first coordinate of $Q = [x_Q^2 : 1]$.
7. The message to be signed is a bit string M and the signature is a pair (x_R^2, s) where x_R^2 is the first coordinate of $R = [x_R^2 : 1]$ and s belongs to $\{0, \ldots, \ell - 1\}$.

The key generation, signature and verification algorithms are shown in Table 14. The explanations of the various operations used in these algorithms are as follows.

1. The function $\mathsf{first}_i(\cdot)$ returns the first i bytes of the input and the function $\mathsf{last}_j(\cdot)$ returns the last j bytes of the input.
2. The function clamp has been defined in Section 3.1.
3. For any point P on $\mathcal{K}_{a^2,b^2}$ and a scalar n, the function $\mathsf{scalarMultProj}(P, n)$ is a modification of the function $\mathsf{scalarMult}(P, n)$ shown in Table 11. The modification consists of returning $[U(\theta) : V(\theta)]$ in Step 16 (instead of $U(\theta)/V(\theta)$). The function $\mathsf{scalarMultFBProj}(P, n)$ is defined to be a similar modification of the function $\mathsf{scalarMultFB}(P, n)$.

Note that the scalars d', r and h are outputs of the clamp function, while s is not. Since s is computed as $s = (r - hd') \bmod \ell$, it is not possible to ensure that it is the output of the clamp function. Consequently, the number of iterations required by the Montgomery ladder in a scalar multiplication by s is determined by the position of the most significant bit of s which is equal to 1. Note that this is a feature of qDSA and does not arise due to instantiation of qDSA by a Kummer line. On the other hand, since s is public, constant time scalar multiplication by s is not an issue.

keyGen()	sign$((d', d''), x_Q^2, M)$	verify$(x_Q^2, M, (x_R^2, s))$
1. $d = \mathsf{random}(\{0,1\}^{256})$;	1. $r = \mathsf{clamp}(\mathsf{first}_{\mathfrak{b}}(H(d''\|\|M)))$;	1. if $(x_Q^2 = \perp)$ or $(x_R^2 = \perp)$ then
2. $\mathsf{str} = H(d)$;	2. $x_R^2 = \mathsf{scalarMultFB}(P, r)$;	2. return F;
3. $d' = \mathsf{clamp}(\mathsf{first}_{\mathfrak{b}}(\mathsf{str}))$;	3. $h = \mathsf{clamp}(\mathsf{first}_{\mathfrak{b}}(H(x_R^2\|\|x_Q^2\|\|M)))$;	3. $Q = [x_Q^2 : 1]$; $R = [x_R^2 : 1]$;
4. $d'' = \mathsf{last}_{64-\mathfrak{b}}(\mathsf{str})$;	4. $s = (r - hd') \bmod \ell$;	4. $h = \mathsf{clamp}(\mathsf{first}_{\mathfrak{b}}(H(x_R^2\|\|x_Q^2\|\|M)))$;
5. $x_Q^2 = \mathsf{scalarMultFB}(P, d')$;	5. return (x_R^2, s).	5. $T_0 = \mathsf{scalarMultFBProj}(P, s)$;
6. return $((d', d''), x_Q^2)$.		6. $T_1 = \mathsf{scalarMultProj}(Q, h)$;
		8. $\mathbf{T}' = ((s+1) \bmod 2)\mathbf{T})$;
		9. return $\mathsf{Check}_{L,\mu}(\psi(T_0), \psi(T_1), \psi(R) + \mathbf{T}')$;

Table 14. Key generation, signing and verification algorithms on Kummer line.

The verification algorithm should return $\mathsf{Check}_{K,a^2,b^2}(T_0, T_1, R)$. Step 9 of the verification algorithm, however, returns $\mathsf{Check}_{L,\mu}(\psi(T_0), \psi(T_1), \psi(R) + \mathbf{T}')$. Below, we argue that this is correct.

Note $T_0 = sP$, $T_1 = hQ = hd'P$ and $R = rP$. Using (18) we have

$$\begin{aligned} \psi(T_0) = \psi(sP) &= s\psi(P) + (s + 1 \bmod 2)\mathbf{T}; \\ \psi(T_1) = \psi(hd'P) &= hd'\psi(P) + (hd' + 1 \bmod 2)\mathbf{T}; \\ \psi(R) = \psi(rP) &= r\psi(P) + (r + 1 \bmod 2)\mathbf{T}. \end{aligned}$$

So,

$$\begin{aligned} \psi(T_0) \pm \psi(T_1) = s\psi(P) \pm hd'\psi(P) + (s \pm hd' \bmod 2)\mathbf{T}; \\ \psi(R) = r\psi(P) + (r + 1 \bmod 2)\mathbf{T}. \end{aligned} \tag{27}$$

From (26), $\mathsf{Check}_{K,a^2,b^2}(T_0, T_1, R)$ returns T if and only if

"either $\mathsf{Check}_{L,\mu}(\psi(T_0), \psi(T_1), \psi(R))$ returns T, or, $\mathsf{Check}_{L,\mu}(\psi(T_0), \psi(T_1), \psi(R) + \mathbf{T})$ returns T"

Recall that the Check predicate returns T if and only if the x-coordinate of the third component is equal to either the x-coordinate of the sum or the x-coordinate of the difference of the first two components. Using this along with (27) and $r = s + hd' \bmod \ell$, we obtain that the above condition is equivalent to

$$\text{“}\mathsf{Check}_{L,\mu}(\psi(T_0), \psi(T_1), \psi(R) + (s \pm hd' + r + 1 \bmod 2)\mathbf{T}) \text{ returns } \mathsf{T}.\text{”} \tag{28}$$

The scalars h, d' and r are all outputs of the clamp function and so they are all even integers. As a result, (28) simplifies to the following condition which is used in Step 9 of the verification algorithm.

$$\text{“}\mathsf{Check}_{L,\mu}(\psi(T_0), \psi(T_1), \psi(R) + (s + 1 \bmod 2)\mathbf{T}) \text{ returns } \mathsf{T}.\text{”} \tag{29}$$

Since $\mathbf{T}' = (s + 1 \bmod 2)\mathbf{T}$, we have that Step 9 of the verification algorithm correctly returns the required value of $\mathsf{Check}_{K,a^2,b^2}(T_0, T_1, R)$.

We have implemented qDSA for all the Kummer lines presented in this paper and the code is publicly available from [26]. Table 15 presents the timing results for key generation, signing and verification of 32-byte messages corresponding to the three Kummer lines.

Remarks:

algorithm	key gen	sign	verify
qDSA-KL2519(81,20)	84302	91422	153916
qDSA-KL25519(82,77)	100614	128882	205674
qDSA-KL2663(260,139)	100198	127448	200650

Table 15. Timing results (number of cycles) on Skylake for qDSA implemented on the Kummer lines presented in this work.

1. In [50], timings for qDSA implementations have been reported for 8-bit AVR ATmega and 32-bit ARM Cortex M0 platforms. Implementations of qDSA labelled "C ref" are available as part of the code for [50] and [35]. We have measured the timings of these implementations on Skylake. The number of required cycles for these implementations turn out to be several times higher than the figures reported in Table 15. Since it is not proper to compare a reference implementation with an optimised implementation, we do not report the timings of the "C ref" code for [50] and [35].
2. On the AVR ATmega and the ARM Cortex M0 platforms, for qDSA, [50] reports faster speed of signing and verification compared to Ed25519 [6]. Timings for qDSA on Intel platforms have not been reported in [50, 35]. To obtain an idea of how qDSA compares to Ed25519 on Skylake, we have timed the signing algorithm of Ed25519 on Skylake and it takes about 50,000 cycles. Comparing to the timings in Table 15 shows that Ed25519 is faster on Skylake. This is possibly due to the utilisation of pre-computed tables which is possible on Skylake but, not on smaller processors.
3. The work [35] considers various Kummer lines and provides explicit connections between them. We refer to this work for a good understanding of the general picture of the relationships between various Kummer lines. The Kummer line considered in this work is called the squared Kummer line in [35]. Among the general results proved in [35] is an involution from a squared Kummer line to the Kummer line arising from a Montgomery curve. It is mentioned that this involution can be composed with the Check operation for Montgomery curves to define a check operation on a squared Kummer line. Our approach of defining the Check operation for Legendre curve given by (24) and (25), and the simplification to (29) do not appear in [35].

9 Conclusion

This work has shown that compared to existing proposals, Kummer line based scalar multiplication for genus one curves over prime order fields offers competitive performance using SIMD operations. Previous works on implementation of Kummer arithmetic had focused completely on genus two. By showing competitive implementation also in genus one, our work fills a gap in the existing literature.

Acknowledgement

We would like to thank Pierrick Gaudry for helpful comments and clarifying certain confusion regarding conversion from Kummer line to elliptic curve. We would also like to thank Peter Schwabe for clarifying certain implementation issues regarding Curve25519 and Kummer surface computation in genus 2. Thanks to Alfred Menezes, René Struik, Patrick Longa, the reviewers of Asiacrypt 2017, and the reviewers of the present paper for comments.

References

1. J. Barwise and P. Eklof. Lefschetz's principle. *Journal of Algebra*, 13(4):554–570, 1969.
2. D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography - PKC*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006.

3. D. J. Bernstein. Elliptic vs. hyperelliptic, part I. *Talk at ECC*, 2006.
4. D. J. Bernstein, C. Chuengsatiansup, T. Lange, and P. Schwabe. Kummer strikes back: New DH speed records. In *Advances in Cryptology - ASIACRYPT*, volume 8873 of *Lecture Notes in Computer Science*, pages 317–337. Springer, 2014.
5. D. J. Bernstein and T. Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. `http://safecurves.cr.yp.to/index.html`, accessed 15th September, 2016.
6. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2011.
7. Daniel J. Bernstein and Peter Schwabe. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2012.
8. Guido Bertoni and Jean-Sébastien Coron, editors. *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*. Springer, 2013.
9. Joppe W. Bos, Craig Costello, Hüseyin Hisil, and Kristin E. Lauter. Fast cryptography in genus 2. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 194–210. Springer, 2013.
10. Joppe W. Bos, Craig Costello, Hüseyin Hisil, and Kristin E. Lauter. High-performance scalar multiplication using 8-dimensional GLV/GLS decomposition. In Bertoni and Coron [8], pages 331–348.
11. Brainpool. ECC standard. `http://www.ecc-brainpool.org/ecc-standard.htm`.
12. 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.
13. Tung Chou. Sandy2x: New Curve25519 speed records. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, volume 9566 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2015.
14. Ping Ngai Chung, Craig Costello, and Benjamin Smith. Fast, uniform scalar multiplication for genus 2 Jacobians with fast Kummers. In *Selected Areas in Cryptography*, 2016. to appear.
15. R. Cosset. Factorization with genus 2 curves. *Mathematics of Computation*, 79(270):1191–1208, 2010.
16. C. Costello and P. Longa. Four($\mathbb{Q}$): Four-dimensional decompositions on a $\mathbb{Q}$-curve over the Mersenne prime. In *Advances in Cryptology - ASIACRYPT Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 214–235. Springer, 2015.
17. 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.
18. Neil Costigan and Peter Schwabe. Fast elliptic-curve cryptography on the cell broadband engine. In Bart Preneel, editor, *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, volume 5580 of *Lecture Notes in Computer Science*, pages 368–385. Springer, 2009.
19. Curve25519. Wikipedia page on Curve25519. `https://en.wikipedia.org/wiki/Curve25519`, accessed 15th September, 2016.
20. M. J. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report, National Institute of Standards and Technology (NIST), 2015. `http://www.nist.gov/manuscript-publication-search.cfm?pub_id=919061`.
21. Armando Faz-Hernández, Patrick Longa, and Ana H. Sánchez. Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves. In Josh Benaloh, editor, *Topics in Cryptology - CT-RSA 2014 - The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings*, volume 8366 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2014.
22. Armando Faz-Hernández and Julio López. Fast implementation of Curve25519 using AVX2. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, volume 9230 of *Lecture Notes in Computer Science*, pages 329–345. Springer, 2015.
23. E.V. Flynn. Formulas for Kummer on genus 2. `http://people.maths.ox.ac.uk/flynn/genus2/kummer/`, accessed on 15th September, 2016.

24. E.V. Flynn. The group law on the Jacobian of a curve of genus 2. *J. reine angew. Math.*, 439:45–69, 1993.
25. Code for Kummer Line Computations. `https://github.com/skarati/KummerLineV02`.
26. Code for qDSA on Kummer Line. `https://github.com/skarati/qDSA`.
27. G. Frey and H.-G. Rück. The strong Lefschetz principle in algebraic geometry. *Manuscripta Mathematica*, 55(3):385–401, 1986.
28. P. Gaudry. Fast genus 2 arithmetic based on theta functions. *J. Mathematical Cryptology*, 1(3):243–265, 2007.
29. P. Gaudry. Personal communication, 2016.
30. P. Gaudry and D. Lubicz. The arithmetic of characteristic 2 Kummer surfaces and of elliptic Kummer lines. *Finite Fields and Their Applications*, 15(2):246–260, 2009.
31. P. Gaudry and É. Schost. Genus 2 point counting over prime fields. *J. Symb. Comput.*, 47(4):368–400, 2012.
32. S. Gueron. Software optimizations for cryptographic primitives on general purpose x86_64 platforms. *Tutorial at IndoCrypt*, 2011.
33. Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptographic Engineering*, 5(2):141–151, 2015.
34. Darrel Hankerson, Koray Karabina, and Alfred Menezes. Analyzing the Galbraith-Lin-Scott point multiplication method for elliptic curves over binary fields. *IEEE Trans. Computers*, 58(10):1411–1420, 2009.
35. Huseyin Hisil and Joost Renes. On kummer lines with full rational 2-torsion and their usage in cryptography. Cryptology ePrint Archive, Report 2018/839, 2018. `https://eprint.iacr.org/2018/839`.
36. Jun ichi Igusa. *Theta functions.* Springer, 1972.
37. Sabyasachi Karati and Palash Sarkar. Kummer for genus one over prime order fields. In Takagi and Peyrin [53], pages 3–32.
38. Neal Koblitz. Elliptic curve cryptosystems. *Math. Comp.*, 48(177):203–209, 1987.
39. Neal Koblitz. Hyperelliptic cryptosystems. *J. Cryptology*, 1(3):139–150, 1989.
40. 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.
41. Patrick Longa and Francesco Sica. Four-dimensional Gallant-Lambert-Vanstone scalar multiplication. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 718–739. Springer, 2012.
42. Victor S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology - CRYPTO'85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, pages 417–426. Springer Berlin Heidelberg, 1985.
43. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.
44. Peter L. Montgomery. Five, six, and seven-term karatsuba-like formulae. *IEEE Trans. Computers*, 54(3):362–369, 2005.
45. D. Mumford. *Tata lectures on theta I.* Progress in Mathematics 28. Birkh äuser, 1983.
46. U.S. Department of Commerce/National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS-186-3. `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`, 2009.
47. 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.
48. Thomaz Oliveira, Julio López, Diego F. Aranha, and Francisco Rodríguez-Henríquez. Lambda coordinates for binary elliptic curves. In Bertoni and Coron [8], pages 311–330.
49. Thomaz Oliveira, Julio López, and Francisco Rodríguez-Henríquez. Software implementation of Koblitz curves over quadratic fields. 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*, volume 9813 of *Lecture Notes in Computer Science*, pages 259–279. Springer, 2016.
50. Joost Renes and Benjamin Smith. qdsa: Small and secure digital signatures with curve-based diffie-hellman key pairs. In Takagi and Peyrin [53], pages 273–302.
51. Certicom Research. SEC 2: Recommended elliptic curve domain parameters. `http://www.secg.org/sec2-v2.pdf`, 2010.
52. Nigel P. Smart and Samir Siksek. A fast diffie-hellman protocol in genus 2. *J. Cryptology*, 12(1):67–73, 1999.
53. Tsuyoshi Takagi and Thomas Peyrin, editors. *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, volume 10625 of *Lecture Notes in Computer Science*. Springer, 2017.
54. NUMS: Nothing up my sleeve. `https://tools.ietf.org/html/draft-black-tls-numscurves-00`.

A Proofs of Theta Identities

For the sake of completeness, we provide the proofs of (2), (3) and (4). These are derived from the definition of theta functions with characteristics given by (1).

A.1 Proof of (2)

We can write $\vartheta[\xi_1, \xi_2](-w, \tau)$ as

$$\vartheta[\xi_1, \xi_2](-w, \tau) = \sum_{n \in \mathbb{Z}} \exp\left[\pi i (n+\xi_1)^2 \tau + 2\pi i (n+\xi_1)(-w+\xi_2)\right] \tag{30}$$

Let $m = -n - 2\xi_1$. As n runs through all the integers and $\xi_1 \in \{0, \frac{1}{2}\}$, m will also run over all the integers. Using $m + \xi_1 = -n - \xi_1$ in (30) we have

$$\begin{aligned}
\vartheta[\xi_1, \xi_2](-w, \tau) &= \sum_{m \in \mathbb{Z}} \exp\left[\pi i (m+\xi_1)^2 \tau + 2\pi i (-m-\xi_1)(-w+\xi_2)\right] \\
&= \sum_{m \in \mathbb{Z}} \exp\left[\pi i (m+\xi_1)^2 \tau + 2\pi i (m+\xi_1)(w-\xi_2)\right] \\
&= \sum_{m \in \mathbb{Z}} \exp\left[\pi i (m+\xi_1)^2 \tau + 2\pi i (m+\xi_1)(w+\xi_2) - 4\pi i (m+\xi_1)\xi_2\right] \\
&= \sum_{m \in \mathbb{Z}} \exp\left[\pi i (m+\xi_1)^2 \tau + 2\pi i (m+\xi_1)(w+\xi_2)\right] \exp(-4\pi i m \xi_2) \exp(-4\pi i \xi_1 \xi_2)
\end{aligned}$$

Now $4\xi_2$ is either zero or an even integer and m is an integer and so $\exp(-4\pi i m \xi_2) = 1$. Similarly, $4\xi_1\xi_2$ is also zero or one. Therefore, $\exp(-4\pi i \xi_1 \xi_2) = \exp(-\pi i)^{4\xi_1\xi_2} = (-1)^{4\xi_1\xi_2}$. This shows

$$\vartheta[\xi_1, \xi_2](-w, \tau) = (-1)^{4\xi_1\xi_2} \sum_{m \in \mathbb{Z}} \exp\left[\pi i (m+\xi_1)^2 \tau + 2\pi i (m+\xi_1)(w+\xi_2)\right]$$

which proves (2).

A.2 Proof of (3)

We can write the $\vartheta_1(w)$, $\vartheta_2(w)$, $\Theta_1(w)$ and $\Theta_2(w)$ as given below:

$$\vartheta_1(w) = \vartheta[0, 0](w, \tau) = \sum_{n \in \mathbb{Z}} \exp\left[\pi i n^2 \tau + 2\pi i n w\right] \tag{31}$$

$$\vartheta_2(w) = \vartheta\left[0, 1/2\right](w, \tau) = \sum_{n \in \mathbb{Z}} \exp\left[\pi i n^2 \tau + \pi i n + 2\pi i n w\right] \tag{32}$$

$$\Theta_1(w) = \vartheta[0, 0](w, 2\tau) = \sum_{n \in \mathbb{Z}} \exp\left[2\pi i n^2 \tau + 2\pi i n w\right] \tag{33}$$

$$\Theta_2(w) = \vartheta\left[1/2, 0\right](w, 2\tau) = \sum_{n \in \mathbb{Z}} \exp\left[2\pi i (n+1/2)^2 \tau + 2\pi i (n+1/2) w\right]. \tag{34}$$

Let w_1 and w_2 be in $\mathbb{C}$. From (33), we can write

$$\begin{aligned}
&\Theta_1(w_1+w_2)\Theta_1(w_1-w_2)\\
&=\left(\sum_{n_1\in\mathbb{Z}}\exp\left[2\pi i n_1^2\tau+2\pi i n_1(w_1+w_2)\right]\right)\left(\sum_{n_2\in\mathbb{Z}}\exp\left[2\pi i n_2^2\tau+2\pi i n_2(w_1-w_2)\right]\right)\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[2\pi i n_1^2\tau+2\pi i n_1(w_1+w_2)+2\pi i n_2^2\tau+2\pi i n_2(w_1-w_2)\right]\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[2\pi i n_1^2\tau+2\pi i n_2^2\tau+2\pi i(n_1+n_2)w_1+2\pi i(n_1-n_2)w_2\right]\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\begin{bmatrix}\pi i n_1^2\tau+\pi i n_2^2\tau+2\pi i n_1 n_2\tau++2\pi i(n_1+n_2)w_1+\\ \pi i n_1^2\tau+\pi i n_2^2\tau-2\pi i n_1 n_2\tau+2\pi i(n_1-n_2)w_2\end{bmatrix}\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i(n_1+n_2)^2\tau+\pi i(n_1-n_2)^2\tau+2\pi i(n_1+n_2)w_1+2\pi i(n_1-n_2)w_2)\right].
\end{aligned}$$

Let $m_1=n_1+n_2$ and $m_2=n_1-n_2$ so that $m_1+m_2=2n_1$ is even. We obtain

$$\Theta_1(w_1+w_2)\Theta_1(w_1-w_2)=\sum_{\substack{m_1\in\mathbb{Z}\\ m_1+m_2=even}}\sum_{m_2\in\mathbb{Z}}\exp\left[\pi i m_1^2\tau+\pi i m_2^2\tau+2\pi i m_1 w_1+2\pi i m_2 w_2)\right]. \tag{35}$$

From (31), we can write

$$\begin{aligned}
&\vartheta_1(w_1)\vartheta_1(w_2)\\
&=\left(\sum_{n_1\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+2\pi i n_1 w_1\right]\right)\left(\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_2^2\tau+2\pi i n_2 w_2\right]\right)\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+2\pi i n_1 w_1+\pi i n_2^2\tau+2\pi i n_2 w_2\right].
\end{aligned}\tag{36}$$

Similarly, from (32), we have

$$\begin{aligned}
&\vartheta_2(w_1)\vartheta_2(w_2)\\
&=\left(\sum_{n_1\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+\pi i n_1+2\pi i n_1 w_1\right]\right)\left(\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_2^2\tau+\pi i n_2+2\pi i n_2 w_2\right]\right)\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+\pi i n_1+2\pi i n_1 w_1+\pi i n_2^2\tau+\pi i n_2+2\pi i n_2 w_2\right].
\end{aligned}\tag{37}$$

Adding (36) and (37), we get

$$\begin{aligned}
&\vartheta_1(w_1)\vartheta_1(w_2)+\vartheta_2(w_1)\vartheta_2(w_2)\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+2\pi i n_1 w_1+\pi i n_2^2\tau+2\pi i n_2 w_2\right]\\
&+\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+\pi i n_1+2\pi i n_1 w_1+\pi i n_2^2\tau+\pi i n_2+2\pi i n_2 w_2\right]\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+2\pi i n_1 w_1+\pi i n_2^2\tau+2\pi i n_2 w_2\right](1+\exp(\pi i(n_1+n_2)))\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+2\pi i n_1 w_1+\pi i n_2^2\tau+2\pi i n_2 w_2\right]\left(1+(-1)^{(n_1+n_2)}\right)
\end{aligned}$$

If $n_1 + n_2$ is odd, the corresponding terms in the series will vanish, whereas if $n_1 + n_2$ is even, there will be a factor of 2. Therefore

$$\begin{aligned}\vartheta_1(w_1)\vartheta_1(w_2) + \vartheta_2(w_1)\vartheta_2(w_2) &= 2\sum_{\substack{n_1\in\mathbb{Z}\\ n_1+n_2=even}}\sum_{n_2\in\mathbb{Z}} \exp\left[\pi i n_1^2\tau + 2\pi i n_1 w_1 + \pi i n_2^2\tau + 2\pi i n_2 w_2\right] \\ &= 2\Theta_1(w_1+w_2)\Theta_1(w_1-w_2) \text{ (using (35))}. \end{aligned} \tag{38}$$

From (34), we can write

$$\begin{aligned}&\Theta_2(w_1+w_2)\Theta_2(w_1-w_2) \\ &= \left(\sum_{n_1\in\mathbb{Z}} \exp\left[2\pi i\left(n_1+\frac{1}{2}\right)^2\tau + 2\pi i\left(n_1+\frac{1}{2}\right)(w_1+w_2)\right]\right) \\ &\quad\times\left(\sum_{n_2\in\mathbb{Z}} \exp\left[2\pi i\left(n_2+\frac{1}{2}\right)^2\tau + 2\pi i\left(n_2+\frac{1}{2}\right)(w_1-w_2)\right]\right) \\ &= \sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}} \exp\begin{bmatrix} 2\pi i\left(n_1+\frac{1}{2}\right)^2\tau + 2\pi i\left(n_1+\frac{1}{2}\right)(w_1+w_2) \\ +2\pi i\left(n_2+\frac{1}{2}\right)^2\tau + 2\pi i\left(n_2+\frac{1}{2}\right)(w_1-w_2)\end{bmatrix} \\ &= \sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}} \exp\left[2\pi i\left(n_1+\frac{1}{2}\right)^2\tau + 2\pi i\left(n_2+\frac{1}{2}\right)^2\tau + 2\pi i(n_1+n_2+1)w_1 + 2\pi i(n_1-n_2)w_2\right] \\ &= \sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}} \exp\begin{bmatrix} \pi i\left(n_1+\frac{1}{2}\right)^2\tau + 2\pi i\left(n_1+\frac{1}{2}\right)\left(n_2+\frac{1}{2}\right)\tau + \pi i\left(n_2+\frac{1}{2}\right)^2\tau + \\ \pi i\left(n_1+\frac{1}{2}\right)^2\tau - 2\pi i\left(n_1+\frac{1}{2}\right)\left(n_2+\frac{1}{2}\right)\tau + \pi i\left(n_2+\frac{1}{2}\right)^2\tau + \\ 2\pi i(n_1+n_2+1)w_1 + 2\pi i(n_1-n_2)w_2\end{bmatrix} \\ &= \sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}} \exp\left[\pi i(n_1+n_2+1)^2\tau + 2\pi i(n_1+n_2+1)w_1 + \pi i(n_1-n_2)^2\tau + 2\pi i(n_1-n_2)w_2\right]\end{aligned}$$

Let $m_1 = n_1 + n_2 + 1$ and $m_2 = n_1 - n_2$ so that $m_1 + m_2 = 2n_1 + 1$. So, we have

$$\Theta_2(w_1+w_2)\Theta_2(w_1-w_2) = \sum_{\substack{m_1\in\mathbb{Z}\\ m_1+m_2=odd}}\sum_{m_2\in\mathbb{Z}} \exp\left[\pi i m_1^2\tau + \pi i m_2^2\tau + 2\pi i m_1 w_1 + 2\pi i m_2 w_2)\right]. \tag{39}$$

Subtracting (37) from (36) and simplifying gives

$$\vartheta_1(w_1)\vartheta_1(w_2) - \vartheta_2(w_1)\vartheta_2(w_2) = 2\sum_{\substack{n_1\in\mathbb{Z}\\ n_1+n_2=odd}}\sum_{n_2\in\mathbb{Z}} \exp\left[\pi i n_1^2\tau + 2\pi i n_1 w_1 + \pi i n_2^2\tau + 2\pi i n_2 w_2\right] \tag{40}$$

$$= 2\Theta_2(w_1+w_2)\Theta_2(w_1-w_2) \text{ (using (39))}. \tag{41}$$

This completes the proof of (3).

A.3 Proof of (4)

From (31), it can be written that

$$\begin{aligned}
&\vartheta_1(w_1+w_2)\vartheta_1(w_1-w_2)\\
&=\left(\sum_{n_1\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+2\pi i n_1(w_1+w_2)\right]\right)\left(\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_2^2\tau+2\pi i n_2(w_1-w_2)\right]\right)\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+2\pi i n_1(w_1+w_2)+\pi i n_2^2\tau+2\pi i n_2(w_1-w_2)\right]\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+2\pi i(n_1+n_2)w_1+\pi i n_2^2\tau+2\pi i(n_1-n_2)w_2\right].
\end{aligned}\tag{42}$$

In (42), note that n_1+n_2 and n_1-n_2 are both even or both odd. From (32), it can be written that

$$\begin{aligned}
&\vartheta_2(w_1+w_2)\vartheta_2(w_2-w_2)\\
&=\left(\sum_{n_1\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+\pi i n_1+2\pi i n_1(w_1+w_2)\right]\right)\left(\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_2^2\tau+\pi i n_2+2\pi i n_2(w_1-w_2)\right]\right)\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+\pi i n_1+2\pi i n_1(w_1+w_2)+\pi i n_2^2\tau+\pi i n_2+2\pi i n_2(w_1-w_2)\right]\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+\pi i n_1+2\pi i(n_1+n_2)w_1+\pi i n_2^2\tau+\pi i n_2+2\pi i(n_1-n_2)w_2\right]\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+2\pi i(n_1+n_2)w_1+\pi i n_2^2\tau+2\pi i(n_1-n_2)w_2\right]\exp(\pi i(n_1+n_2))\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+2\pi i(n_1+n_2)w_1+\pi i n_2^2\tau+2\pi i(n_1-n_2)w_2\right](-1)^{(n_1+n_2)}.
\end{aligned}$$

Again n_1+n_2 and n_1-n_2 are both even or both odd. If n_1+n_2 is even, then $\exp(\pi i(n_1+n_2))=1$ and if n_1+n_2 is odd, then $\exp(\pi i(n_1+n_2))=-1$.

From (33), we can write

$$\begin{aligned}
&\Theta_1(2w_1)\Theta_1(2w_2)\\
&=\left(\sum_{n_1\in\mathbb{Z}}\exp\left[2\pi i n_1^2\tau+2\pi i n_1(2w_1)\right]\right)\left(\sum_{n_2\in\mathbb{Z}}\exp\left[2\pi i n_2^2\tau+2\pi i n_2(2w_2)\right]\right)\\
&=\left(\sum_{n_1\in\mathbb{Z}}\exp\left[2\pi i n_1^2\tau+4\pi i n_1 w_1\right]\right)\left(\sum_{n_2\in\mathbb{Z}}\exp\left[2\pi i n_2^2\tau+4\pi i n_2 w_2\right]\right)\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[2\pi i n_1^2\tau+4\pi i n_1 w_1+2\pi i n_2^2\tau+4\pi i n_2 w_2\right]\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i n_1^2\tau+\pi i n_2^2\tau+2\pi i n_1 n_2\tau+4\pi i n_1 w_1+\pi i n_1^2\tau+\pi i n_2^2\tau-2\pi i n_1 n_2\tau+4\pi i n_2 w_2\right]\\
&=\sum_{n_1\in\mathbb{Z}}\sum_{n_2\in\mathbb{Z}}\exp\left[\pi i(n_1+n_2)^2\tau+4\pi i n_1 w_1+\pi i(n_1-n_2)^2\tau+4\pi i n_2 w_2\right].
\end{aligned}$$

Let $m_1' = n_1 + n_2$ and $m_2' = n_1 - n_2$, where $n_1, n_2 \in \mathbb{Z}$. Therefore $m_1' + m_2' = 2n_1$ and $m_1' - m_2' = 2n_2$ are both even. So,

$$\Theta_1(2w_1)\Theta_1(2w_2) = \sum_{\substack{m_1' \in \mathbb{Z} \\ m_1'+m_2' = even}} \sum_{m_2' \in \mathbb{Z}} \exp\left[\pi i m_1'^2 \tau + 2\pi i (m_1' + m_2') w_1 + \pi i m_2'^2 \tau + 2\pi i (m_1' - m_2') w_2\right] \quad (43)$$

From (34), we can write

$$\begin{aligned}
&\Theta_2(2w_1)\Theta_2(2w_2) \\
&= \left(\sum_{n_1 \in \mathbb{Z}} \exp\left[2\pi i \left(n_1 + \frac{1}{2}\right)^2 \tau + 2\pi i \left(n_1 + \frac{1}{2}\right)(2w_1)\right]\right) \\
&\quad \times \left(\sum_{n_2 \in \mathbb{Z}} \exp\left[2\pi i \left(n_2 + \frac{1}{2}\right)^2 \tau + 2\pi i \left(n_2 + \frac{1}{2}\right)(2w_2)\right]\right) \\
&= \left(\sum_{n_1 \in \mathbb{Z}} \exp\left[2\pi i \left(n_1 + \frac{1}{2}\right)^2 \tau + 4\pi i \left(n_1 + \frac{1}{2}\right) w_1\right]\right) \\
&\quad \times \left(\sum_{n_2 \in \mathbb{Z}} exp\left[2\pi i \left(n_2 + \frac{1}{2}\right)^2 \tau + 4\pi i \left(n_2 + \frac{1}{2}\right) w_2\right]\right) \\
&= \sum_{n_1 \in \mathbb{Z}} \sum_{n_2 \in \mathbb{Z}} \exp\left[2\pi i \left(n_1 + \frac{1}{2}\right)^2 \tau + 4\pi i \left(n_1 + \frac{1}{2}\right) w_1 + 2\pi i \left(n_2 + \frac{1}{2}\right)^2 \tau + 4\pi i \left(n_2 + \frac{1}{2}\right) w_2\right] \\
&= \sum_{n_1 \in \mathbb{Z}} \sum_{n_2 \in \mathbb{Z}} \exp\begin{bmatrix} \pi i \left(n_1 + \frac{1}{2}\right)^2 \tau + \pi i \left(n_2 + \frac{1}{2}\right)^2 \tau + 2\pi i \left(n_1 + \frac{1}{2}\right)\left(n_2 + \frac{1}{2}\right)\tau + 4\pi i \left(n_1 + \frac{1}{2}\right) w_1 \\ \pi i \left(n_1 + \frac{1}{2}\right)^2 \tau + \pi i \left(n_2 + \frac{1}{2}\right)^2 \tau - 2\pi i \left(n_1 + \frac{1}{2}\right)\left(n_2 + \frac{1}{2}\right)\tau + 4\pi i \left(n_2 + \frac{1}{2}\right) w_2 \end{bmatrix} \\
&= \sum_{n_1 \in \mathbb{Z}} \sum_{n_2 \in \mathbb{Z}} \exp\left[\pi i (n_1 + n_2 + 1)^2 \tau + 4\pi i \left(n_1 + \frac{1}{2}\right) w_1 + \pi i (n_1 - n_2)^2 \tau + 4\pi i \left(n_2 + \frac{1}{2}\right) w_2\right].
\end{aligned}$$

Let $m_1' = n_1 + n_2 + 1$ and $m_2' = n_1 - n_2$ and so $m_1' + m_2'$ and $m_1' - m_2'$ are both odd. We can write

$$\Theta_2(2w_1)\Theta_2(2w_2) = \sum_{\substack{m_1' \in \mathbb{Z} \\ m_1'+m_2' = odd}} \sum_{m_2' \in \mathbb{Z}} \exp\left[\pi i m_1'^2 \tau + 2\pi i (m_1' + m_2') w_1 + \pi i m_2'^2 \tau + 2\pi i (m_1' - m_2') w_2\right] \quad (44)$$

Addition of (43) and (44) creates a series where m_1' and m_2' varies over $\mathbb{Z}$ and $m_1' + m_2'$ and $m_1' - m_2'$ are either both odd or both even. Therefore the series is exactly the same as the series defined by (42) and we get the identity

$$\vartheta_1(w_1 + w_2)\vartheta_1(w_1 - w_2) = \Theta_1(2w_1)\Theta_1(2w_2) + \Theta_2(2w_1)\Theta_2(2w_2)$$

Similarly, if we subtract (44) from (43), we get the exact series defined by (43). This gives the identity

$$\vartheta_2(w_1 + w_2)\vartheta_2(w_1 - w_2) = \Theta_1(2w_1)\Theta_1(2w_2) - \Theta_2(2w_1)\Theta_2(2w_2)$$

which completes the proof of (4).