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 had suggested the use of the associated Kummer line to speed up scalar multiplication. In this work, we explore this idea in details. The first task is to obtain an elliptic curve in the 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 Kummer ladder supports parallelism and can be implemented very efficiently in constant time using the single-instruction multiple-data (SIMD) operations available in modern processors. We report implementation using Intel intrinsics and the code is publicly available. The timing results show that the performance of the Kummer line based approach compare favourably with previously proposed genus one curves over prime order fields.

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 work of Koblitz [37] and Miller [40] introducing elliptic curves and the work of Koblitz [38] 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. Necessary conditions have been identified for a curve to be considered secure. The currently known necessary conditions can be found at [5]. These necessary conditions, however, are not known to be sufficient. The security of curve-based cryptography is based on the conjecture that the discrete logarithm problem in the relevant group is computationally difficult. This conjecture does not hold in the setting of quantum computing. Since the actual realisation of quantum computers seem to be quite some time away, it is still meaningful to conduct research for identifying new classes of secure and efficient curves.

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 and Curve25519. Both of these 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 [18]. 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 curve) have concrete proposals in this setting. Bicoins use the `sec256k1` curve. See [5] for a further listing of such primes and curves. We further note that prime order fields are considered

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

important. It has been mentioned in [2] that prime order fields "have the virtue of minimizing the number of security concerns for elliptic-curve cryptography." 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.

While genus one curves over prime order fields continue to be of enduring interest (as evidenced by [43, 18, 11, 47, 13]), in recent years, the research on efficient implementation [39, 9, 10, 44, 20, 17, 16, 45] has largely focussed on either genus 2, or, composite order fields (binary extension, prime-squared) with the goal of improving efficiency. The use of endomorphisms for fields of large characteristic is based on [27, 26] while for binary fields, the base-ϕ expansion of the scalar originates in the work of Koblitz. The setting of genus one over prime order fields received some recent focus in the implementation of the NIST P-256 curve [32] and through [46] which presented complete addition formulas for prime order curves in the short Weierstrass form over finite fields of characteristic not equal to 2 or 3.

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

Reference	genus	form	field order	endomorphisms
NIST P-256 [43]	1	Weierstrass	prime	no
Curve25519 [2]	1	Montgomery	prime	no
Brainpool [11]	1	Weierstrass	prime	no
NUMS [47]	1	twisted Edwards	prime	no
secp256k1 [13]	1	Weierstrass	prime	no
Longa-Sica [39]	1	twisted Edwards	p^2	yes
Bos et al. [9]	2	Kummer	prime	yes
Bos et al. [10]	2	Kummer	p^2	yes
Oliviera et al. [44]	1	Weierstrass/Koblitz	2^n	yes
Faz-Hernández et al. [20]	1	twisted Edwards	p^2	yes
Costello et al. [17]	1	Montgomery	p^2	yes
Bernstein et al. [4]	2	Kummer	prime	no
Costello et al. [16]	1	twisted Edwards	p^2	yes
Oliviera et al. [45]	1	Weierstrass/Koblitz	2^n	yes
This work	1	Kummer	prime	no

Our Contributions

In this work, we go back to the setting of genus one curves over a prime order field. Two efficient models of curves that have been considered in genus one are the Montgomery [41] and the (twisted) Edwards model [19, 6]. An issue of central importance is to be able to perform scalar multiplications in constant time. The Montgomery form supports a ladder based scalar multiplication which ensures constant time execution. Using unified formula for point addition leads to constant time scalar multiplication for Edwards form curve.

The contribution of this paper is to propose a new curve for the setting of genus one and prime order field. Actual computation is done over the Kummer line associated with the 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 provide a competitive alternative to known curves in the setting of genus one and prime order fields.

Like in the case of Montgomery curve, 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 ensures that the resulting code is constant time. We describe and implement a vectorised version of the laddering algorithm which is also constant time. Our target is the 128-bit security level. The work consists of several aspects.

Choice of the underlying field: We work over the field $\mathbb{F}_p$ where p is the prime $2^{251} - 9$. This prime has been earlier suggested in [1]. The approach of multi-limb representation [2] of field elements is adopted. We argue that using 9 limbs to represent elements of $\mathbb{F}_p$ suffices to ensure efficient arithmetic. Alternative choices for the underlying field, such as $\mathbb{F}_{2^{255}-19}$ and $\mathbb{F}_{(2^{127}-1)^2}$ have been considered and we argue that compared to $\mathbb{F}_p$ these would lead to slower field arithmetic.

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 a Kummer line with $a^2 = 101$ and $b^2 = 61$ for which the corresponding elliptic curve satisfies the necessary conditions listed in [5]. The Kummer line also has a base point $[x : z]$ with $x^2 = 4$ and $z^2 = 1$. The small values of the parameters and the base point are of great advantage in improving the efficiency of scalar multiplication. Further details about Kummer line are provided later.

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. Such instructions have been used for implementing field arithmetic over $\mathbb{F}_{2^{127}-1}$. 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 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 constant time, 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, Montgomery ladder 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 choice of the Kummer line ensures 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 Kummer ladder vis-a-vis the Montgomery ladder.

An example of the Kummer ladder is shown in Figure 1. 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 Kummer ladder 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.

In contrast, the Montgomery ladder is shown in Figure 2 which has been reproduced from [2]. The structure of this ladder is not as regular as the Kummer ladder. This makes it difficult to optimally group together the multiplications for SIMD implementation. More importantly, it does not seem easy to work only on the packed representation within a ladder. AVX2 based implementation of Curve25519 has been reported in [21]. This work, though, could group together only 2 multiplications/squarings. At a forum[3], Tung Chou comments that it would better to find 4 independent multiplications/squarings and vectorise them. For the Kummer ladder shown in Figure 1 performing vectorisation of 4 independent multiplications/squarings comes quite naturally. This shows that the Kummer ladder is better suited for SIMD implementation than the Montgomery ladder.

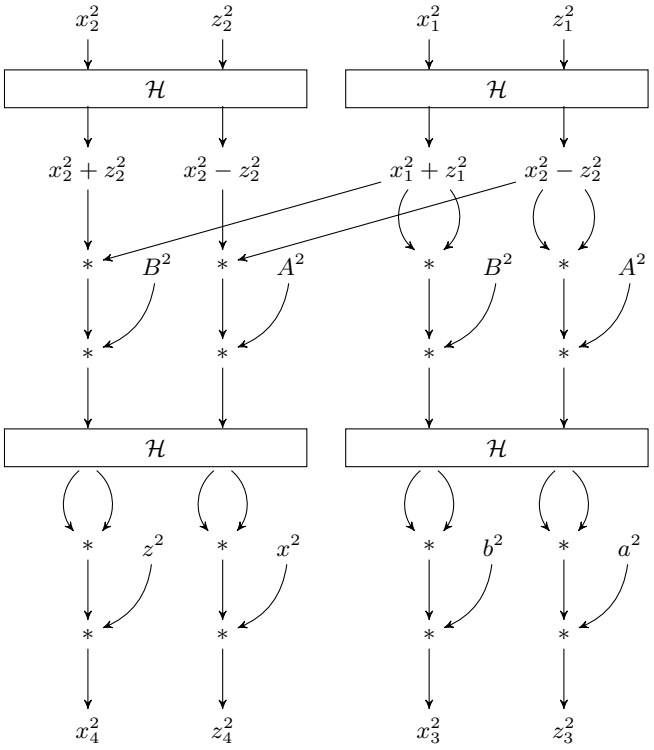

Fig. 1. Kummer ladder.

Another choice for implementation is the twisted Edwards form. Using explicit unified formulas for addition leads to constant time scalar multiplication. A non-adjacent form (NAF) representation of the scalar will require 1 doubling and 1/3 additions per bit. Using operation counts from [6] this comes to 6 field mulitplications, 4.33 squarings and 0.33 multiplications by constants per bit; using faster explicit formula from [33] requires 10.33 field multiplications and 1.33 multiplications by constants. These operation counts are higher than either Kummer or Montgomery. Working with a windowed NAF method along with a pre-computed table can provide substantial speed-up in the fixed base scalar

[3] `https://moderncrypto.org/mail-archive/curves/2015/000637.html`

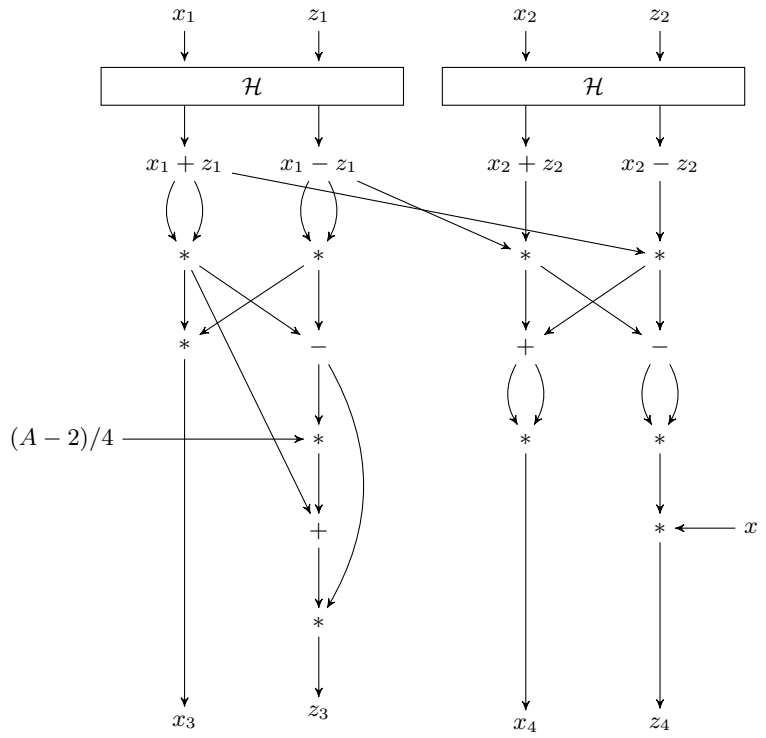

Fig. 2. Montgomery ladder.

multiplication. See [14] for details of this approach for Curve25519 and [32] for NIST P-256. For this work, we focus only on ladder based algorithms which do not require a pre-computed table and works for both fixed base and variable base scalar multiplications.

Implementation: We report an implementation of the particular Kummer line over $\mathbb{F}_{2^{251}-9}$ on the Haswell processor of Intel using AVX2 instructions. The implementation has been carried out using Intel intrinsics. For variable base scalar multiplication, the timing that we obtain is competitive with the reported timings for other genus one curves over prime order fields, specifically Curve25519 and NIST P-256.

2 Background

There is a deep theory about Kummer varieties. We require only some very rudimentary facts which are mentioned below. For details of the theory, we refer to [42, 34]. Cryptographic applications 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 [42, 23, 22] for arithmetic on Kummer surface associated to genus two curves.

Let $\mathbb{F}_q$ be the finite field consisting of q elements having characteristic not equal to two.

2.1 Theta Functions

Let $\vartheta_1, \vartheta_2, \Theta_1, \Theta_2 : \mathbb{F}_q \to \mathbb{F}_q$ be functions such that the following identities hold:

$$\begin{aligned} 2\Theta_1(w_1+w_2)\Theta_1(w_1-w_2) &= \vartheta_1(w_1)\vartheta_1(w_2) + \vartheta_2(w_1)\vartheta_2(w_2); \\ 2\Theta_2(w_1+w_2)\Theta_2(w_1-w_2) &= \vartheta_1(w_1)\vartheta_1(w_2) - \vartheta_2(w_1)\vartheta_2(w_2); \end{aligned} \tag{1}$$

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

Putting $w_1 = w_2 = w$, we obtain

$$\begin{aligned}2\Theta_1(2w)\Theta_1(0) &= \vartheta_1(w)^2+\vartheta_2(w)^2;\\ 2\Theta_2(2w)\Theta_2(0) &= \vartheta_1(w)^2-\vartheta_2(w)^2;\end{aligned} \tag{3}$$

$$\begin{aligned}\vartheta_1(2w)\vartheta_1(0) &= \Theta_1(2w)^2+\Theta_2(2w)^2;\\ \vartheta_2(2w)\vartheta_2(0) &= \Theta_1(2w)^2-\Theta_2(2w)^2.\end{aligned} \tag{4}$$

Putting $w = 0$ in (3), we obtain

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

Let

$$a = \vartheta_1(0),\ b = \vartheta_2(0),\ A = \Theta_1(0) \text{ and } B = \Theta_2(0).$$

Then from (5) we obtain

$$A^2 = (a^2+b^2)/2 \text{ and } B^2 = (a^2-b^2)/2.$$

Assume that the values a and b are known and hence the values of A^2 and B^2 are also known. Let $\mathbb{P}^1(\mathbb{F}_q)$ denote the projective line over $\mathbb{F}_q$. Consider the map

$$\varphi : \mathbb{F}_q \to \mathbb{P}^1(\mathbb{F}_q) \text{ given by } \varphi(w) = [\vartheta_1(w) : \vartheta_2(w)]. \tag{6}$$

Suppose that $\varphi(w) = [\vartheta_1(w) : \vartheta_2(w)]$ is known for some $w \in \mathbb{F}_q$. Using (3) it is possible to compute $\Theta_1(2w)$ and $\Theta_2(2w)$ and then using (4) 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 (3), it is possible to obtain $\Theta_1(2w_1), \Theta_1(2w_2), \Theta_2(2w_1)$ and $\Theta_2(2w_2)$. Then (2) 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.

Remark: We have not provided the definitions of the functions $\vartheta_1, \vartheta_2, \Theta_1$ and Θ_2. As a result the identities given in (1) and (2) appear to come out of nowhere. There are, in fact, explicit definitions of ϑ_1 and ϑ_2 as maps from $\mathbb{C}$ to $\mathbb{C}$ from which the functions Θ_1 and Θ_2 are defined. Given these definitions, the identities in (1) and (2) can be proved to hold over $\mathbb{C}$. One can refer to the Lefschetz principle to argue that the identities also hold over $\mathbb{F}_q$. For the present work, only the identities are required and how they have been obtained are not important.

2.2 Kummer Line

Given $a = \vartheta_1(0) \neq 0$ and $b = \vartheta_2(0) \neq 0$, the image of $\mathbb{F}_q$ under φ is the Kummer line $\mathcal{K}_{a,b}$. The task of computing $\varphi(2w)$ from $\varphi(w)$ is called doubling in $\mathcal{K}_{a,b}$ 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 addition in $\mathcal{K}_{a,b}$. Doubling and differential addition are completely defined by the values of a and b.

For a point $R = \varphi(w)$ in $\mathcal{K}_{a,b}$, let $\mathsf{dbl}(R)$ denote the point $\varphi(2w) \in \mathcal{K}_{a,b}$. For points $R_1 = \varphi(w_1)$, $R_2 = \varphi(w_2)$ and $S = \varphi(w_1-w_2)$, let $\mathsf{diffAdd}(R_1, R_2, S)$ denote the point $\varphi(w_1+w_2)$. The discussion given above shows that it is possible to compute $\mathsf{dbl}(R)$ and $\mathsf{diffAdd}(R_1, R_2, S)$.

Let $P = \varphi(w)$ be a point on $\mathcal{K}_{a,b}$ and $n \geq 1$ be a positive integer. The scalar multiplication of P by n, denoted as nP, is the point $\varphi(nw) \in \mathcal{K}_{a,b}$. Using doubling and pseudo-addition, it is possible to compute nP from P without actually knowing w. This is done using the ladder algorithm shown in Table 2. The input to the first ladder step are the squared coordinates of $(P, 2P)$. Suppose, at the i-th

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 $S =$ dbl(P); for $i = \ell - 2, \ell - 3, \ldots, 0$ do $(R, S) =$ ladder(R, S, n_i); return R.	ladder(R, S, b) if $(b = 0)$ $S =$ diffAdd(R, S, P); $R =$ dbl(R); else $R =$ diffAdd(R, S, P); $S =$ dbl(S); return (R, S).

Table 2. Scalar multiplication using a ladder.

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 $((2k+1)P, (2k+2)P)$.

2.3 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 $(a, b) \in E$, $\sigma(a, b) = (a, -b)$. The Kummer line associated with E is the image of E/σ under the map φ defined in (6). In fact, the map φ provides an embedding of E/σ into $\mathbb{P}^1(\mathbb{F}_q)$.

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

$$E_\mu : Y^2 = X(X-1)(X-\mu) \tag{7}$$

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

$$\mu = \frac{a^4}{a^4 - b^4}. \tag{8}$$

An explicit map $\psi : \mathcal{K}_{a,b} \to E_\mu/\sigma$ has been given in [30]. The corrected version [29] of this map is as follows. Let $[x : \pm z]$ represent the two points $[x : z]$ and $[x : -z]$ on $\mathcal{K}_{a,b}$ associated to E_μ. These two points map to the same point on E_μ.

$$\psi([x : \pm z]) = \begin{cases} \infty & \text{if } [x : z] = [b : \pm a]; \\ \left(\frac{a^2x^2}{a^2x^2 - b^2z^2}, \ldots\right) & \text{otherwise.} \end{cases} \tag{9}$$

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 $\mathcal{K}_{a,b}$. Since ψ maps two elements of $\mathcal{K}_{a,b}$ to a single point of E_μ/σ, the inverse map ψ^{-1} does not uniquely determine the preimage. Let $\mathbf{P} \in E_\mu/\sigma$. Then either $\mathbf{P} = (X, \ldots)$ or $\mathbf{P} = \infty$. Then

$$\psi^{-1}(\mathbf{P}) = \begin{cases} [b : \pm a] & \text{if } \mathbf{P} = \infty; \\ \left[\sqrt{\frac{b^2X}{a^2(X-1)}} : \pm 1\right] & \text{if } \mathbf{P} = (X, \ldots) \text{ and } X \neq 1; \\ [1 : 0] & \text{if } \mathbf{P} = (X, \ldots) \text{ and } X = 1. \end{cases} \tag{10}$$

Notation: We will use upper-case bold face letters to denote points of E_μ and upper case normal letters to denote points of $\mathcal{K}_{a,b}$.

Consistency of scalar multiplication: Let $\mathcal{K}_{a,b}$ and E_μ be such that (8) holds. Suppose that P is a point on $\mathcal{K}_{a,b}$. It is possible to compute nP using the laddering algorithm described in Section 2.2. Using ψ it is possible to map P to a point in E_μ/σ and perform scalar multiplication in E_μ. Consistency means that doing the computation in $\mathcal{K}_{a,b}$ and in E_μ should give rise to the same result. The map ψ by itself does not guarantee this. It is required to compose ψ with addition by a point of order two in E_μ. The details are as follows.

Let $\mathbf{T}$ be a point of order 2 on E_μ. Let $\psi(P) = \mathbf{P}$; $\mathbf{Q} = \mathbf{P} + \mathbf{T}$; $\mathbf{Q}_n = n\mathbf{Q}$; $P_n = nP$; $\psi(P_n) = \mathbf{P}_n$; $\mathbf{P}_n + \mathbf{T} = \mathbf{Q}'_n$. Then $\mathbf{Q}_n = \mathbf{Q}'_n$. Here $n\mathbf{Q}$ denotes scalar multiplication in E_μ.

Conversely, let $\mathbf{Q}$ be a point on E_μ; $\mathbf{Q}_n = n\mathbf{Q}$; $\mathbf{P} = \mathbf{Q} - \mathbf{T}$; $P = \psi^{-1}(\mathbf{P})$; $P_n = nP$; $\mathbf{P}_n = \mathbf{Q}_n - \mathbf{T}$; $P'_n = \psi^{-1}(\mathbf{P}_n)$. Note that ψ^{-1} does not return a unique element of $\mathcal{K}_{a,b}$ and one chooses the sign arbitrarily. As a result, P_n and P'_n are equal only up to the sign. Since $\mathbf{T}$ is a point of order 2, the operation $-\mathbf{T}$ is the same as the operation $+\mathbf{T}$.

Figure 3 depicts the above statements in pictorial form. Formal proofs of the above statements are very messy to obtain. On the other hand, using the explicit formulas, it is easy to verify these relations using a software such as Magma.

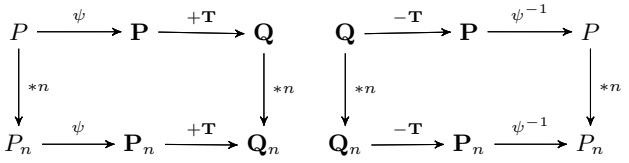

Fig. 3. Consistency of scalar multiplications on E_μ and $\mathcal{K}_{a,b}$.

The points $(0,0)$, $(1,0)$ and $(\mu,0)$ are the three points of order 2 on E_μ. For concreteness, we fix $\mathbf{T} = (0,0)$.

Relation between the discrete logarithm problems: Suppose the Kummer line $\mathcal{K}_{a,b}$ is chosen such that the corresponding curve E_μ has a cyclic subgroup $\mathfrak{G}$ which can be used for cryptographic purposes. So, in particular, the order of $\mathfrak{G}$ is a prime. Let $\mathfrak{G} = \langle \mathbf{P} \rangle$.

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 $\mathcal{K}_{a,b}$. Map the point $\mathbf{P}$ (resp. $\mathbf{Q}$) to $P \in \mathcal{K}_{a,b}$ (resp. $Q \in \mathcal{K}_{a,b}$) using translation by $\mathbf{T}$ followed by ψ^{-1} as described above. 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 $\mathcal{K}_{a,b}$. 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 $\mathcal{K}_{a,b}$ will be equally hard. This fact forms the basis for using Kummer line for cryptographic applications.

2.4 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 a similar calculation in [41, 12]. For the genus two case, this problem has been addressed in [15].

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

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

3 Square Only Setting

Let $P = [x : z]$ be a point in $\mathcal{K}_{a,b}$. Doubling computes the point $2P = [x_3 : z_3]$. Suppose $w \in \mathbb{F}_q$ such that $x = \vartheta_1(w)$, $z = \vartheta_2(w)$, $x_3 = \vartheta_1(2w)$ and $z_3 = \vartheta_2(2w)$. As discussed in Section 2.1, computation of x_3 and z_3 can be done using the theta identities. Looking at these identities carefully, one notices that the computation would be more uniform if x_3^2 and z_3^2 are computed from x^2 and z^2. We provide the details. For the case of genus two, this approach was advocated in [9, 3].

Using (3), we obtain

$$\Theta_1(2w)^2 = \frac{(x^2 + z^2)^2}{4A^2};\ \Theta_2(2w)^2 = \frac{(x^2 - z^2)^2}{4B^2}.$$

Then from (4)

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

For $[x : z] \in \mathcal{P}^1(\mathcal{F}_q)$, $[x : z] = [\lambda x : \lambda z]$ for any non-zero λ. Using this, we have

$$\begin{aligned}
[x_3'^2 : z_3'^2] &= \left[\frac{(\Theta_1(2w)^2 + \Theta_2(2w)^2)^2}{a^2} : \frac{(\Theta_1(2w)^2 - \Theta_2(2w)^2)^2}{b^2}\right] \\
&= \left[b^2(\Theta_1(2w)^2 + \Theta_2(2w)^2)^2 : a^2(\Theta_1(2w)^2 - \Theta_2(2w)^2)^2\right] \\
&= \left[b^2\left(\frac{(x^2+z^2)^2}{4A^2} + \frac{(x^2-z^2)^2}{4B^2}\right)^2 : a^2\left(\frac{(x^2+z^2)^2}{4A^2} - \frac{(x^2-z^2)^2}{4B^2}\right)^2\right] \\
&= \left[b^2\left(B^2(x^2+z^2)^2 + A^2(x^2-z^2)^2\right)^2 : a^2\left(B^2(x^2+z^2)^2 - A^2(x^2-z^2)^2\right)^2\right] \\
&= [x_3^2 : z_3^2].
\end{aligned}$$

So, it is sufficient to compute $[x_3^2 : z_3^2]$. This computation is shown as Algorithm sdbl in Table 3. The requirement is to double the point $P = [x : z]$ and obtain the point $2P = [x_3 : z_3]$. Instead the above procedure takes input (x^2, z^2) and returns (x_3^2, z_3^2).

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

Table 3. Double and differential addition in the square-only setting.

For differential addition, one starts with points $P_1 = [x_1 : z_1]$, $P_2 = [x_2 : z_2]$ and the difference $P_1 - P_2 = P = [x : z]$ in $\mathcal{K}_{a,b}$ and computes $P_1 + P_2 = [x_3 : z_3]$. The above idea of square only computation extends to differential addition. Given $x_1^2, z_1^2, x_2^2, z_2^2, x^2$ and z^2, the computation of x_3^2 and z_3^2 is shown as Algorithm $\mathsf{sdiffAdd}$ in Table 3.

Suppose $P = [x_1 : z_1]$ and the requirement is to compute $[x_n : z_n]$ where $nP = [x_n : z_n]$. The square only approach starts with x_1^2, z_1^2 and computes x_n^2 and z_n^2. This is achieved by replacing dbl and $\mathsf{diffAdd}$ in Algorithm $\mathsf{scalarMult}$ with the sdbl and $\mathsf{sdiffAdd}$ respectively.

Let the ℓ-bit binary expansion of n be $n = (1, n_{\ell-2}, \ldots, n_0)$. Algorithm $\mathsf{scalarMult}$ 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. For $u, v \in \mathbb{F}_q$, the Hadamard transform $\mathcal{H}(u, v)$ is defined to be $(u + v, u - v)$. 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 the (x_3^2, z_3^2) and (x_4^2, z_4^2), where $(x_3^2, z_3^2) = \mathsf{sdbl}(x_1^2, z_1^2)$ and $(x_4^2, z_4^2) = \mathsf{sdiffAdd}(x_1^2, z_1^2, x_2^2, z_2^2)$.

Redefinition of ψ and ψ^{-1} In the square only setting the pair of inputs (x^2, z^2) represents the two points $[x : \pm z] \in \mathbb{P}^1(\mathbb{F}_q)$. We redefine the maps ψ and ψ^{-1} to reflect this change. In particular, the redefined ψ takes as input (x^2, z^2) and returns the x-coordinate of the corresponding point on E_μ while the redefined $\psi^{-1}(\mathbf{P})$ returns (x^2, z^2) where $[x : \pm z]$ are the points in $\mathbb{P}^1(\mathbb{F}_q)$ corresponding to $\mathbf{P}$.

$$\psi(x^2, z^2) = \begin{cases} \infty & \text{if } (x^2, z^2) = (b^2, a^2); \\ \left(\frac{a^2x^2}{a^2x^2 - b^2z^2}, \ldots\right) & \text{otherwise.} \end{cases} \tag{11}$$

$$\psi^{-1}(\mathbf{P}) = \begin{cases} (b^2, a^2) & \text{if } \mathbf{P} = \infty; \\ \left(\frac{b^2X}{a^2(X-1)}, 1\right) & \text{if } \mathbf{P} = (X, \ldots) \text{ and } X \neq 1; \\ (1, 0) & \text{if } \mathbf{P} = (X, \ldots) \text{ and } X = 1. \end{cases} \tag{12}$$

3.1 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 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 $\psi(P) + \mathbf{T} = (X_P, \ldots)$. The point P is the base point on $\mathcal{K}_{a^2,b^2}$.

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 $\mathbf{Q} = \psi(Q) + \mathbf{T}$ and $\mathbf{R} = \psi(R) + \mathbf{T}$, where as before $T = (0, 0)$ is a point of order 2 on E_μ. The consistency of scalar multiplication have been described in Section 2.3. This extends easily to the square only setting. By the consistency of scalar multiplication, we have $\mathbf{Q} = n\mathbf{P}$.

Consider $\mathbf{Q} = \psi(Q) + \mathbf{T}$. Let $\alpha_Q = a^2x_Q^2$ and $\beta_Q = a^2x_Q^2 - b^2z_Q^2$ 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

$$\begin{aligned}\gamma_Q &= \mu\beta_Q = \mu(a^2x_Q^2 - b^2z_Q^2);\\ \delta_Q &= \alpha_Q = a^2x_Q^2.\end{aligned}$$

Similarly, we obtain $\mathbf{R} = (X_R, \ldots)$ where $X_R = \gamma_R/\delta_R$ and γ_R, δ_R given by the above 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 2.4. This gives

$$\begin{aligned}Y_Q &= \frac{-1}{2Y_P}\left((X_Q - X_P)^2(X_P + X_Q + X_R - \mu - 1) - f(X_Q) - Y_P^2\right)\\ &= -\frac{1}{2Y_P}\left(\left(\frac{\gamma_Q}{\delta_Q} - X_P\right)^2\left(X_P + \frac{\gamma_Q}{\delta_Q} + \frac{\gamma_R}{\delta_R} - \mu - 1\right) - f\left(\frac{\gamma_Q}{\delta_Q}\right) - Y_P^2\right)\\ &= \frac{-1}{2Y_P\delta_Q^3\delta_R}\left((\gamma_Q - X_P\delta_Q)^2\left((X_P - \mu - 1)\delta_Q\delta_R + \gamma_Q\delta_R + \gamma_R\delta_Q\right) - \gamma_Q\delta_R(\gamma_Q - \delta_Q)(\gamma_Q - \mu\delta_Q) - Y_P^2\delta_Q^3\delta_R\right).\end{aligned}$$

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.

4 Choice of the Kummer Line

We will work in the square only setting. This means we will only be interested in the values a^2 and b^2 and the values a and b will not be required at all. In view of this, for the rest of the paper, we will abuse notation and write $\mathcal{K}_{a^2,b^2}$ to denote the Kummer line where $\vartheta_1(0) = a$ and $\vartheta_2(0) = b$. Extending to points, we will only require the squared coordinates of the points. So, if $[x : z]$ is the base point, we will require x^2 and z^2.

Let p be the prime $2^{251} - 9$. This prime has been earlier suggested for use in elliptic curve cryptology in [1]. Below we mention some advantages of using this prime.

Let $a, b \in \mathbb{Z}_p$ be such that $a^2 = 101 \bmod p$ and $b^2 = 61 \bmod p$. We consider the Kummer line $\mathcal{K}_{101,61}$ in the square-only setting. The other two important theta constants are $A^2 = (a^2 + b^2)/2 = 81 \bmod p$ and $B^2 = (a^2 - b^2)/2 = 20 \bmod p$.

Let E_μ be the Legendre form curve on $\mathbb{F}_p$ defined by the following equation.

$$E_\mu : \; y^2 = x(x - 1)(x - \mu) \tag{13}$$

where,

$$\mu = 1206726007146220574413275938482709147718539915424620268563212958338639434566.$$

The Kummer Line $\mathcal{K}_{101,61}$ corresponds to the curve E_μ, where it is easy to verify that $\mu = \frac{101^2}{101^2-61^2} \bmod p$. The point $P = (x^2, z^2) = (4, 1)$ is on $\mathcal{K}_{101,61}$ and for fixed-base scalar multiplications, we have used this point as the base point. Let $\mathbf{P}_1$ and $\mathbf{P}_2$ be the two points $\psi(P) + \mathbf{T}$ with $\mathbf{T} = (0, 0)$. Then

$\mathbf{P}_1 = (1938104715086878786971252875284064217697418978000925866319190826077664422729,$
$330859156190535243962560325472066076258362513382957892990840829189134441973 3)$
$\mathbf{P}_2 = (1938104715086878786971252875284064217697418978000925866319190826077664422729,$
$309911226760778667360990026800836357831061886971688696324641208355940881506).$

Any one of these can be used as a base point for scalar multiplication in E_μ.

4.1 Procedure for Selecting $\mathcal{K}_{101,61}$

Our target was 128-bit security level. The selection procedure for the $\mathcal{K}_{101,61}$ was an exhaustive search over certain range of parameters. The search terminates when an underlying Legendre form curve appropriate conjectured security is obtained. The actual search procedure is the following.

Start with $\alpha = 1$ and increase α by one at each iteration; vary β from 1 to α; set $a^2 = \alpha$ and $b^2 = \beta$ and check whether the corresponding Kummer line $\mathcal{K}_{a^2,b^2}$ has the desired conjectured security level. This checking was done on the associated curve E_μ. The choice $(a^2 = 101, b^2 = 61)$ is the first pair of values for which the desired security parameters are achieved.

The checking of the security of the curve E_μ associated with the Kummer line $\mathcal{K}_{a^2,b^2}$ has been done based on the criterion provided by [5]. The relevant security parameters associated with the $\mathcal{K}_{101,61}$ are provided in Table 4. Based on Table 4 and the recommendations in [5], we conclude that $\mathcal{K}_{101,61}$ is a

Table 4. Security Parameters of the Kummer Line $\mathcal{K}_{101,61}$

ℓ (sz of max prime subgroup)	4523128485832663883733241601901871400546714350314447\ 7905323951655758463005 3
cofactor	8
rho security	123.83 bits
embedding degree $= \frac{\ell-1}{7}$	6461612122618091262476059431288387715066734786163 49\ 68436177073793940661436
complex multiplication field discriminant ($\approx 2^{246.3}$)	$-139594264541126456366502846787211587344503522447547$\ 43385315147073016860219100 $= -1 \times 59 \times 83 \times 15948101111 \times 178742772691495830118613$\ 51122757543356377061226686249959424 73
ℓ' (sz of max prime subgroup of the non-trivial twist)	4523128485832663883733241601901871400490003201688721\ 27505022858504236695257
cofactor of the non-trivial twist	8
twist rho security	123.83 bits
twist embedding degree $= \ell' - 1$	4523128485832663883733241601901871400490003201688721\ 27505022858504236695256
joint rho security	122.33 bits

good choice for 128-bit security level.

4.2 Appropriateness of the Prime $2^{251} - 9$

The prime p is a generalised Mersenne prime of the form $2^{251} + \delta$ with $\delta = -9$. The elements of $\mathbb{F}_p$ can be represented using 9 limbs so that all arithmetic can be carried out without any overflow. This issue is explained below in more details.

One alternative prime is $2^{255} - 19$ which has been used in the widely popular Curve25519. For this prime, however, a 10-limb representation is required to ensure no-overflow arithmetic. The increase from 9 limbs to 10 limbs will significantly degrade the performance of field multiplication and squaring and hence of the scalar multiplication.

In our experiments, using other primes (such as the NIST primes), we were able to find a few other Kummer lines having proper security parameters, though we did not find any suitable Kummer line for the prime $2^{255} - 19$. For all these other Kummer lines, however, the line parameters and the coordinates of the base point are large. So, we did not consider them.

5 Field Arithmetic

For the target Kummer line $\mathcal{K}_{101,61}$, the underlying field is $\mathbb{F}_p$ with $p = 2^{251} - 9$. All Kummer line operations are ultimately built from field operations. So, it is important to have fast field arithmetic to ensure fast Kummer line computation. We use the approach of limb-wise representation and arithmetic for elements of $\mathbb{F}_p$. This approach has been earlier used in [2, 4]. In the rest of this section, we provide the details of the approach for the specific prime that we consider.

5.1 Representation of Field Elements

A general element r of $\mathbb{F}_p$ with $p = 2^{251} - 9$ requires 251 bits to be represented. Consider these 251 bits to be split into 9 parts as $(r_8, r_7, r_6, r_5, r_4, r_3, r_2, r_1, r_0)$, where for $0 \leq i \leq 7$, r_i is 28 bits long and r_8 is 27 bits long. Following previous terminology, we call each part to be a limb. So, a general element r of $\mathbb{F}_p$ can be written as follows:

$$r = r_8 \cdot 2^{224} + r_7 \cdot 2^{196} + r_6 \cdot 2^{168} + r_5 \cdot 2^{140} + r_4 \cdot 2^{112} + r_3 \cdot 2^{84} + r_2 \cdot 2^{56} + r_1 \cdot 2^{28} + r_0. \quad (14)$$

In the above, r is a 251-bit integer and so can be greater than or equal to p. The elements of $\mathbb{F}_p$ are actually obtained as $r \bmod p$. We instead will work directly with 251-bit integers. Doing this gives rise to non-unique representations of the elements $0, \ldots, 8$ in $\mathbb{F}_p$ which are also represented as $2^{251}-9, \ldots, 2^{251}-1$ respectively. While working with "random" 251-bit integers, the probability of encountering one of the integers $2^{251} - 9, \ldots, 2^{251} - 1$ is at most $9/(2^{251} - 9)$ which is negligible. Importantly, working with the non-unique representations does not cause any of the computations to become incorrect. Further, if desired, at the end of all the computations, it is easy to check whether the obtained result is one of $2^{251} - 9, \ldots, 2^{251} - 1$ and if so, to replace it by the corresponding value from $0, \ldots, 8$. In view of this, for the rest of the work, we will work with 251-bit integers as if they are the elements of $\mathbb{F}_p$.

Let

$$\theta = 2^{28}. \quad (15)$$

A polynomial $A(\theta)$ in θ with integer coefficients will be called *a proper polynomial* if $A(\theta)$ is of degree at most 8, i.e., $A(\theta) = a_0 + a_1\theta + \cdots + a_8\theta^8$ and further, $a_0, \ldots, a_7 < 2^{28}$ and $a_8 < 2^{27}$. Clearly, if $A(\theta)$ is a proper polynomial, then $A(\theta)$ represents an element of $\mathbb{F}_p$ in the form (14).

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

$$\begin{array}{l} \mathfrak{P}(\theta) = \sum_{i=0}^{8} \mathfrak{p}_i \theta^i \text{ with} \\ \quad \mathfrak{p}_0 = 2^{28} - 9; \\ \quad \mathfrak{p}_i = 2^{28} - 1; \quad i = 1, \ldots, 7; \text{ and} \\ \quad \mathfrak{p}_8 = 2^{27} - 1. \end{array} \tag{16}$$

The coefficients of a proper polynomial will be represented using 64-bit words. The product of coefficients of two proper polynomials can be computed using a 32-bit multiplication and will also fit in a 64-bit word. In the following sections, we show that all the arithmetic on coefficients that will be required will result in values that fit into 64-bit words. Arithmetic in $\mathbb{F}_p$ will take as input proper polynomials and provide as output the result which is also a proper polynomial.

5.2 Reduction

Using $p = 2^{251} - 9$, we have

$$2^{252} = 2 \times 2^{251} = 2(2^{251} - 9) + 18 \equiv 18 \bmod p.$$

So, multiplying by 2^{252} modulo p is the same as multiplying by 18 modulo p. Recall that we have set $\theta = 2^{28}$ and so $2^{252} = \theta^9$ which implies that $\theta^9 \bmod p = 18$.

Suppose $C(\theta) = \sum_{i=0}^{8} c_i \theta^i$ is a polynomial such that for some $\mathfrak{m} \leq 64$, $c_i < 2^{\mathfrak{m}}$ for all $i = 0, \ldots, 7$. If $\mathfrak{m} > 28$, then $C(\theta)$ is not a proper polynomial. We describe a method to obtain a proper polynomial $D(\theta)$ such that $D(\theta) \equiv C(\theta) \bmod p$. This procedure will be called the reduction step.

The reduction is done iteratively. Let $c_i^{(0)} = c_i$ for $i = 0, \ldots, 8$ and $C^{(0)}(\theta) = \sum_{i=0}^{8} c_i^{(0)} \theta^i$. Set $t_0^{(1)} = 0$. The reduction will successively construction polynomials $C^{(1)}(x), C^{(2)}(x), \ldots$. The construction of $C^{(1)}(x)$ from $C^{(0)}(x)$ will be as follows.

The idea is to iteratively compute $c_0^{(1)}, \ldots, c_7^{(1)} < 2^{28}$, $c_8^{(1)} < 2^{27}$; $t_1^{(1)}, \ldots, t_8^{(1)} < 2^{\mathfrak{m}-28}$ and $t_0^{(2)} < 2^{\mathfrak{m}-27}$ in the following manner. Let $\mathsf{lsb}_i(\cdot)$ denote the i least significant bits of the argument.

- Set $t_0^{(1)} = 0$.
- For $i = 0, \ldots, 7$, let $c_i^{(1)} = \mathsf{lsb}_{28}(c_i^{(0)} + t_i^{(1)})$ and $t_{i+1}^{(1)}$ be such that $c_i^{(0)} + t_i^{(1)} = c_i^{(1)} + 2^{28} t_{i+1}^{(1)}$.
- Let $c_8^{(1)} = \mathsf{lsb}_{27}(c_8^{(0)} + t_8^{(1)})$ and $t_0^{(2)}$ be such that $c_8^{(0)} + t_8^{(1)} = c_8^{(1)} + 2^{27} t_0^{(2)}$.
- Set $c_0^{(1)} \leftarrow c_0^{(1)} + 18 t_0^{(2)}$ and

$$C^{(1)}(\theta) = c_0^{(1)} + c_1^{(1)}\theta + \cdots + c_8^{(1)}\theta^8.$$

The computation can be written out more explicitly in the following manner.

$$\begin{aligned} C^{(0)}(\theta) &= c_0^{(0)} + c_1^{(0)}\theta + \cdots + c_8^{(0)}\theta^8 \\ &= (c_0^{(1)} + t_1^{(1)}\theta) + c_1^{(0)}\theta + \cdots + c_8^{(0)}\theta^8 \\ &= c_0^{(1)} + (t_1^{(1)} + c_1^{(0)})\theta + \cdots + c_8^{(0)}\theta^8 \\ &= c_0^{(1)} + ((c_1^{(1)} + t_2^{(1)}\theta))\theta + \cdots + c_8^{(0)}\theta^8 \\ &= c_0^{(1)} + c_1^{(1)}\theta + (t_2^{(1)} + c_2^{(0)})\theta^2 + \cdots + c_8^{(0)}\theta^8 \\ &\cdots \\ &= c_0^{(1)} + c_1^{(1)}\theta + c_2^{(1)}\theta^2 + \cdots + (t_8^{(1)} + c_8^{(0)})\theta^8 \\ &= c_0^{(1)} + c_1^{(1)}\theta + c_2^{(1)}\theta^2 + \cdots + (c_8^{(1)} + t_0^{(2)}\theta)\theta^8 \end{aligned}$$

$$
\begin{aligned}
&= c_0^{(1)} + c_1^{(1)}\theta + c_2^{(1)}\theta^2 + \cdots + c_8^{(1)}\theta^8 + t_0^{(2)}\theta^9 \\
&\equiv (c_0^{(1)} + 18t_0^{(2)}) + c_1^{(1)}\theta + c_2^{(1)}\theta^2 + \cdots + c_8^{(1)}\theta^8 \pmod{p} \\
&= C^{(1)}(\theta).
\end{aligned}
$$

The last but one step uses $\theta^9 \equiv 18 \bmod p$.

By construction, $t_1^{(1)}, \ldots, t_8^{(1)}$ are all less than $2^{\mathfrak{m}-28}$ and $t_0^{(2)} < 2^{\mathfrak{m}-27}$. In the above, we assume that

$$
t_i^{(1)} + c_i^{(0)} < 2^{64} \quad \text{for} \quad i = 1, \ldots, 8 \text{ and } 2^{28} + 18t_0^{(2)} < 2^{64} \tag{17}
$$

so that the additions can be carried out using 64-bit arithmetic without any overflow. The second condition is required in the computation of $c_0^{(1)} + 18t_0^{(2)}$. Later, in the context where the reduction procedure is applied, we will see that the assumption holds.

In the above computation, $c_0^{(1)}$ is first obtained as the 28 least significant bits of $c_0^{(0)}$ and then is updated as $c_0^{(1)} \leftarrow c_0^{(1)} + 18t_0^{(2)}$. For $C^{(1)}(\theta)$ we have $c_0^{(1)} < 2^{28} + 18 \times 2^{\mathfrak{m}-27}$, $c_1^{(1)}, \ldots, c_7^{(1)} < 2^{28}$ and $c_8^{(1)} < 2^{27}$. If it turns out that $c_0^{(1)}$ is actually less than 2^{28}, then $C^{(1)}(\theta)$ is a proper polynomial and the reduction stops. Otherwise, further reduction is required. A procedure similar to the one used for obtaining $C^{(1)}(\theta)$ from $C^{(0)}(\theta)$ is used to obtain a polynomial $C^{(2)}(\theta)$ from $C^{(1)}(\theta)$. If $C^{(2)}(\theta)$ is proper, then the reduction stops; otherwise, it continues to obtain $C^{(3)}(\theta), \ldots$ until a proper polynomial is obtained. The complete algorithm is described below.

$\mathsf{reduce}(C(\theta))$
input: $C(\theta) = c_0 + c_1\theta + \cdots + c_8\theta^8$, $c_i < 2^{\mathfrak{m}}$, $i = 0, \ldots, 8$;
output: a proper polynomial $D(\theta)$ such that $D(\theta) \equiv C^{(0)}(\theta) \bmod p$;
1. set $C^{(0)}(\theta) = C(\theta)$, i.e., $c_i^{(0)} = c_i$ for $i = 0, \ldots, 8$
1. set $k = 0$;
2. repeat
3. $\quad k \leftarrow k + 1$;
4. $\quad$ set $t_0^{(k)} = 0$;
5. $\quad$ for $i = 0, \ldots, 7$ do
6. $\quad\quad c_i^{(k)} \leftarrow \mathsf{lsb}_{28}(c_i^{(k-1)} + t_i^{(k)})$; let $t_{i+1}^{(k)}$ be such that $c_i^{(k-1)} + t_i^{(k)} = c_i^{(k)} + 2^{28}t_{i+1}^{(k)}$;
7. $\quad$ end for;
8. $\quad c_8^{(k)} \leftarrow \mathsf{lsb}_{27}(c_8^{(k-1)} + t_8^{(k)})$; let $t_0^{(k+1)}$ be such that $c_8^{(k-1)} + t_8^{(k)} = c_8^{(k)} + 2^{27}t_0^{(k+1)}$;
9. $\quad$ set $c_0^{(k)} \leftarrow c_0^{(k)} + 18t_0^{(k+1)}$;
10. $\quad$ set $C^{(k)}(\theta) = c_0^{(k)} + c_1^{(k)}\theta + \cdots + c_8^{(k)}\theta^8$;
11. until $C^{(k)}(\theta)$ is proper;
12. return $C^{(k)}(\theta)$.

The polynomials constructed in Step 10 are successively $C^{(1)}(\theta), C^{(2)}(\theta), \ldots$.

To argue about termination, suppose $\mathfrak{m} = 64$ which is the maximum possible value of $\mathfrak{m}$. (If $\mathfrak{m} < 64$, then termination can happen sooner.) We have already argued that $c_0^{(1)}$ is less than $2^{28} + 18 \times 2^{37} < 2^{42}$. From Step 6, $t_1^{(2)}$ is given by the 14 most significant bits of $c_0^{(1)}$ and so $t_1^{(2)} < 2^{14}$. As a result, $c_1^{(1)} + t_1^{(2)} < 2^{28} + 2^{14} < 2^{29}$. Writing $c_1^{(1)} + t_1^{(2)}$ as $c_1^{(2)} + t_2^{(2)}2^{28}$ where $c_1^{(2)}$ is given by the 28 least significant bits of $c_1^{(1)} + t_1^{(2)}$ shows that $t_2^{(2)} < 2$. We already have that $c_2^{(1)} < 2^{28}$ and so $c_2^{(1)} + t_2^{(2)} \leq 2^{28}$ and as a result $t_3^{(2)} < 2$. Continuing, it can be argued that $t_i^{(2)} < 2$ for $i = 2, \ldots, 8$ and the value of $t_0^{(3)}$ computed in Step 8 is also less than 2.

If for any $i \in \{2, \ldots, 8\}$, $t_i^{(2)}$ turns out to be zero, or, $t_0^{(3)}$ turns out to be zero, then there will be no further carries and the algorithm can immediately stop. The resulting $C^{(2)}(\theta)$ is proper. The only

way in which $t_3^{(2)}$ can be 1 is if $c_2^{(1)} = 2^{28} - 1$; similarly, for $i = 4, \ldots, 8$, the only way in which $t_i^{(2)}$ can be 1 is if $c_{i-1}^{(1)} = 2^{28} - 1$; and the only way in which the value of $t_0^{(3)}$ computed at Step 8 can be 1 is if $c_8^{(1)} = 2^{28} - 1$.

Considering $c_2^{(1)}, \ldots, c_8^{(1)}$ to be random 28-bit integers, the probability of this event is $2^{-28\times 7}$ which is negligible. So, with very high probability, $C^{(2)}(\theta)$ is proper. In fact, with probability $1 - 2^{-28}$, $t_3^{(2)}$ is 0 and the reduction process immediately stops. For a deterministic analysis, it can be argued that if $C^{(2)}(\theta)$ is not proper, then $C^{(3)}(\theta)$ will certainly be proper. Since the chances of considering $C^{(3)}(\theta)$ is very low in practice, we omit the details of this analysis.

5.3 Field Addition

Let $A(\theta) = \sum_{i=0}^{8} a_i\theta^i$ and $B(\theta) = \sum_{i=0}^{8} b_i\theta^i$ be two proper polynomials. Let $C(\theta) = \sum_{i=0}^{8} c_i\theta^i$ where $c_i = a_i + b_i$ for $i = 0, \ldots, 8$. From the bounds on a_i and b_i, we have $c_i < 2^{29} - 1$ for $i = 0, \ldots, 7$ and $c_8 < 2^{28} - 1$. The operation $\mathsf{sum}(A(\theta), B(\theta))$ is defined to be $D(\theta)$ which is obtained as $D(\theta) = \mathsf{reduce}(C(\theta))$. During the computation of $\mathsf{reduce}(\theta)$, the value of $\mathtt{m}$ used in Section 5.2 is 29.

5.4 Field Negation

Let $A(\theta) = \sum_{i=0}^{8} a_i\theta^i$ be a proper polynomial. We wish to compute $-A(\theta)$. By $\mathsf{negate}(A(\theta))$ we denote $T(\theta) = 2\mathfrak{P}(\theta) - A(\theta)$. Reducing $T(\theta)$ modulo p gives the desired answer. Let $T(\theta) = \sum_{i=0}^{8} t_i\theta^i$ so that $t_i = 2\mathfrak{p}_i - a_i$.

From (16), $2^{28} < 2\mathfrak{p}_i < 2^{29}$ for $i = 0, \ldots, 7$ and $2^{27} < 2\mathfrak{p}_8 < 2^{28}$. Since $A(\theta)$ is a proper polynomial, from the bounds on the coefficients of proper polynomials, it follows that $2\mathfrak{p}_i - a_i$ is positive for $i = 0, \ldots, 8$. This eliminates the situation in two's complement subtraction, where the result can be negative. Considering all values to be 64-bit quantities, the computation of t_i is done in the following manner: $t_i = ((2^{64} - 1) - a_i) + (1 + 2\mathfrak{p}_i) \bmod 2^{64}$. The operation $(2^{64} - 1) - a_i$ is equivalent to taking the bitwise complement of b_i which is equivalent to $1^{64} \oplus a_i$.

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

5.5 Multiplication by a Small Constant

Let $A(\theta) = \sum_{i=0}^{8} a_i\theta^i$ be a proper polynomial and $c < 2^{28}$ be a small positive integer. We will require c to be only at most a 16-bit quantity. Note that c is also an element of $\mathbb{F}_p$.

The operation $\mathcal{C}(A(\theta), c)$ will denote the proper polynomial $C(\theta)$ obtained as follows: Compute $B(\theta) = \sum_{i=0}^{8}(ca_i)\theta^i$; and $C(\theta) = \mathsf{reduce}(B(\theta))$.

5.6 Field Multiplication

Field multiplication in $\mathbb{F}_p$ is done by first performing an integer multiplication of two elements of $\mathbb{F}_p$ followed by a reduction modulo p. The 9-limb representation of an element of $\mathbb{F}_p$ can be used to define several strategies for integer multiplication.

Suppose that $A(\theta) = \sum_{i=0}^{8} a_i\theta^i$ and $B(\theta) = \sum_{i=0}^{8} b_i\theta^i$ are proper polynomials representing two elements of $\mathbb{F}_p$. Let $C(\theta)$ be the result of the integer multiplication of $A(\theta)$ and $B(\theta)$. Then $C(\theta)$ can be written as

$$C(\theta) = c_0 + c_1\theta + \cdots + c_{16}\theta^{16} \tag{18}$$

where $c_t = \sum_{s=0}^{t} a_s b_{t-s}$ with the convention that a_i, b_j is zero for $i, j > 8$. For $s = 0, \ldots, 8$, the coefficient $c_{8\pm s}$ is the sum of $(9-s)$ products of the form $a_i b_j$. Since $a_i, b_j < 2^{28}$, it follows that for $s = 0, \ldots, 8$,

$$c_{8\pm s} \leq (9-s)(2^{28}-1)^2 < 2^{60}. \tag{19}$$

In particular, for $t = 0, \ldots, 16$, each c_t fits in a 64-bit word.

The result of the integer multiplication of $A(\theta)$ and $B(\theta)$ in $\mathbb{F}_p$ is $C(\theta)$. There are several strategies for computing $C(\theta)$ from $A(\theta)$ and $B(\theta)$. These are considered below.

Schoolbook: This consists of 81 multiplications of the type $a_i b_j$ with $i, j = 0, \ldots, 8$ and then adding up the relevant products to obtain the c_t's. All the multiplications are independent and can be carried out in parallel. There is no pre-processing involved before performing the multiplications. The post-processing is required to add the desired products.

Karatsuba with 3-way split: Suppose we are required to multiply two polynomials of degrees at most 2 each. This requires a total of 9 multiplications of the coefficients of the polynomials. Using Karatsuba with 3-way split, the number of multiplications can be reduced to 6 at the cost of some extra additions. We provide more details.

Let $a(x) = a_0 + a_1 x + a_2 x^2$ and $b(x) = b_0 + b_1 x + b_2 x^2$ be two quadratic polynomials and we are required to compute $c(x) = a(x)b(x) = \sum_{i=0}^{4} c_i x^i$. The 5 coefficients $c_0, \ldots, c_4$ can be computed using 6 multiplications in the following manner:

$s_1 = (a_0 + a_1)$, $s_2 = (a_1 + a_2)$, $s_3 = (a_0 + a_2)$,
$t_1 = (b_0 + b_1)$, $t_2 = (b_1 + b_2)$, $t_3 = (b_0 + b_2)$,
$m_0 = a_0 b_0$, $m_1 = a_1 b_1$, $m_2 = a_2 b_2$, $m_3 = s_1 t_1$, $m_4 = s_2 t_2$, $m_5 = s_3 t_3$,
$c_0 = m_0$, $c_1 = m_3 - m_0 - m_1$, $c_2 = m_5 - m_0 - m_2 + m_1$, $c_3 = m_4 - m_1 - m_2$, $c_4 = m_2$.

The 6 multiplications are independent and can be carried out in parallel, but, there is some pre-processing required to prepare the inputs to the multiplications and also some post-processing of the outputs of the multiplications to obtain the c_i's.

Consider now the task of multiplying the two polynomials $A(\theta)$ and $B(\theta)$ of degrees at most eight each. The polynomials $A(\theta)$ and $B(\theta)$ can be written as $A(\theta) = A_0(\theta) + A_1(\theta)\theta^3 + A_2(\theta)\theta^6$ and $B(\theta) = B_0(\theta) + B_1(\theta)\theta^3 + B_2(\theta)\theta^6$ where $A_i(\theta)$ and $B_j(\theta)$ are polynomials of degrees at most two. The product is $C(\theta) = A(\theta)B(\theta) = \sum_{i=0}^{4} C_i(\theta)\theta^{3i}$. The $C_i(\theta)$'s can be computed using 6 multiplications of quadratic polynomials in a manner similar to the method described above. Further, each such product of two quadratic polynomials can be computed using 6 multiplications in exactly the manner described above. So, the complete product $A(\theta)B(\theta)$ can be computed using a total of 36 32-bit multiplications. These 36 multiplications themselves can be made independent and computed in parallel, but, this requires substantial amount of pre and post-processing.

Unbalanced Karatsuba: Two 9-coefficient polynomials can be multiplied by breaking each into a 4-coefficient polynomial and a 5-coefficient polynomial. Using Karatsuba at the top level will require the multiplication of one 4-coefficient polynomial and two 5-coefficient polynomials. Again using Karatsuba for these multiplications will require 9 32-bit multiplications for multiplying the 4-coefficient polynomials and 16 32-bit multiplications for multiplying each pair of 5-coefficient polynomials. So, the two 9-coefficient polynomials can be multiplied using a total of 41 32-bit multiplications. The pre and post-processing for this case will be lower than that of Karatsuba with 3-way split, but, the overhead is still quite high.

Hybrid Karatsuba: Suppose $a(x) = a_0 + a_1 x$ and $b(x) = b_0 + b_1 x$ are two polynomials and $c(x) = a(x)b(x) = c_0 + c_1 x + c_2 x^2$ is their product. The basic Karatsuba algorithm computes c_0, c_1 and c_2 using 3 multiplications as $c_0 = a_0 b_0$, $c_2 = a_1 b_1$ and $c_1 = (a_0 + a_1)(b_0 + b_1) - c_0 - c_2$.

Given polynomials $A(\theta)$ and $B(\theta)$ of degrees at most 8 each, write $A(\theta) = a_8\theta^8 + S(\theta)$ and $B(\theta) = b_8\theta^8 + T(\theta)$ where $S(\theta)$ and $T(\theta)$ are of degrees at most 7 each. The product $C(\theta) = A(\theta)B(\theta)$ can be written as

$$C(\theta) = A(\theta)B(\theta) = a_8 b_8 \theta^{16} + (a_8 T(\theta) + b_8 S(\theta))\theta^8 + S(\theta)T(\theta). \tag{20}$$

Computing $C(\theta)$ in this manner requires 17 32-bit multiplications (one to compute $a_8 b_8$; 8 to compute $a_8 T(\theta)$; 8 to compute $b_8 S(\theta)$) plus the multiplications to compute $S(\theta)T(\theta)$.

The product of the two degree 7 polynomials $S(\theta)$ and $T(\theta)$ can be computed using three recursive applications of Karatsuba's algorithm using 27 32-bit multiplications. Write $S(\theta) = S_0(\theta) + S_1(\theta)\theta^4$ and $T(\theta) = T_0(\theta) + T_1(\theta)\theta^4$, where $S_i(\theta)$ and $T_j(\theta)$ are polynomials of degrees at most 4 each. The product

$$\begin{aligned}
&S(\theta)T(\theta) \\
&= S_0(\theta)T_0(\theta) + (S_0(\theta)T_1(\theta) + S_1(\theta)T_0(\theta))\theta^4 + S_1(\theta)T_1(\theta)\theta^8 \\
&= S_0(\theta)T_0(\theta) + ((S_0(\theta) + S_1(\theta))(T_0(\theta) + T_1(\theta)) - S_0(\theta)T_0(\theta) - S_1(\theta)T_1(\theta))\theta^4 + S_1(\theta)T_1(\theta)\theta^8
\end{aligned}$$

can be computed using three multiplications of degree 4 polynomials. Each of the multiplications of degree 4 polynomials can be broken down into 3 multiplications of degree 2 polynomials and further, each of the multiplications of degree 2 polynomials can be computed using 3 multiplications. This leads to a total of 27 multiplications. The actual procedure does not involve any recursive calls; instead, the recursion is completely written out.

So, the total number of 32-bit multiplications required to compute (20) is $17 + 27 = 44$. This number is larger than the 36 32-bit multiplications required using Karatsuba with 3-way split. However, in this case, the amount of pre and post-processing involved is less and consequently in our implementation, this strategy results in the fastest time.

No overflows. The coefficients of $A(\theta)$ and $B(\theta)$ are multiplied using 32-bit arithmetic and the rest of the operations on the coefficients are done using 64-bit arithmetic. We argue that this does not cause any overflow.

Suppose that $C(\theta)$ as given by (18) has been computed, i.e., $c_0, \ldots, c_{16}$ are available.

$$\begin{aligned}
C(\theta) &= (c_{16}\theta^7 + c_{15}\theta^6 + \cdots + c_{10}\theta + c_9)\theta^9 + c_8\theta^8 + \cdots + c_1\theta + c_0 \\
&\equiv 18(c_{16}\theta^7 + c_{15}\theta^6 + \cdots + c_{10}\theta + c_9) + c_8\theta^8 + \cdots + c_1\theta + c_0 \bmod p \\
&\equiv c_8\theta^8 + (c_7 + 18c_{16})\theta^7 + \cdots + (c_1 + 18c_{10})\theta + (c_0 + 18c_9) \bmod p.
\end{aligned}$$

The multiplications by 18 can be computed using 3 left shifts followed by an addition and another left shift.

Let $c_8^{(0)} = c_8$ and for $i = 0, \ldots, 7$, let $c_i^{(0)} = c_i + 18c_{9+i}$. Define

$$C^{(0)}(\theta) = c_0^{(0)} + c_1^{(0)}\theta + \cdots + c_8^{(0)}\theta^8. \tag{21}$$

Using (19), the following bounds on $c_i^{(0)}$ can be obtained.

$$\begin{aligned}
&c_8^{(0)} \le 2^{60}; \\
&c_i^{(0)} \le ((i+1) + 18(9-i-1))(2^{28}-1)^2 \le 145(2^{28}-1)^2 < 2^{64}\ i = 0, \ldots, 7.
\end{aligned} \tag{22}$$

In particular, $c_0^{(0)}, \ldots, c_8^{(0)}$ all fit into 64-bit words. So, the reduction procedure of Section 5.2 is applied to $C^{(0)}(\theta)$. We need to argue that (17) holds. By construction $t_1^{(1)} < 2^{36}$. Proceeding by induction, for $i = 1, \ldots, 7$, $c_i^{(0)} \leq 145(2^{28}-1)^2$ implying $t_i^{(1)} + c_i^{(0)} < 145(2^{28}-1)^2 + 2^{36} < 2^{64}$ and $t_2^{(1)}, \ldots, t_8^{(1)} < 2^{36}$. Since $c_8^{(0)} < 2^{60}$ and $t_8^{(1)} < 2^{36}$, $t_8^{(1)} + c_8^{(0)} < 2^{61}$ so that $t_0^{(2)} < 2^{34}$ (since $c_8^{(1)}$ equals $\mathsf{lsb}_{27}(t_8^{(1)} + c_8^{(0)})$). As a result, $c_0^{(1)} + 18t_0^{(2)} < 2^{39}$. So, all computations required in the reduction step can be carried out using 64-bit additions without any overfull.

Since $c_0^{(1)} < 2^{39}$, $t_1^{(2)}$ is given by the 11 most significant bits of $c_0^{(1)}$ and so $t_1^{(2)} < 2^{11}$. The termination analysis in Section 5.2 assumed $t_1^{(2)} < 2^{14}$. The smaller value of $t_1^{(2)}$ can make the termination possibly a little faster, but, otherwise does not affect anything else.

Given two proper polynomials $A(\theta)$ and $B(\theta)$, by $\mathcal{M}(A(\theta), B(\theta))$ we denote the proper polynomial $D(\theta)$ which is obtained as $D(\theta) = \mathsf{reduce}(C^{(0)}(\theta))$ and $C^{(0)}(\theta)$ is as defined in (21).

5.7 Field Squaring

Let $A(\theta)$ be a proper polynomial representing an element of $\mathbb{F}_p$ as described in (5.1). We wish to obtain a proper polynomial $C(\theta)$ such that $C(\theta) \equiv A^2(\theta) \bmod p$. As in the case of multiplication, this is done in two steps. First $C(\theta) = A^2(\theta)$ is computed considering θ to be a place-holder variable and second, $C(\theta)$ is reduced modulo p to obtain a proper polynomial $D(\theta)$. The reduction step is the one described in Section (5.2) and the overflow analysis is similar to that for multiplication. So, we only need to consider the procedure for computing $C(\theta) = A^2(\theta)$.

Most computer architectures do not have different instructions for 32-bit multiplication and 32-bit squaring. So, there is no difference in the times for 32-bit multiplications and 32-bit squarings. Accordingly, we will also not distinguish between these times.

The strategy for multiplying using Karatsuba with 3-way split described above can be directly used for squaring $A(\theta)$ and requires 36 32-bit multiplications. The hybrid Karatsuba strategy, however, can be made more efficient for squaring as we next describe.

Write $A(\theta) = a_8\theta^8 + S(\theta)$. Then

$$A^2(\theta) = a_8^2\theta^{16} + 2a_8S(\theta) + S^2(\theta). \tag{23}$$

One 32-bit multiplication is required to compute a_8^2 and 8 32-bit multiplications are required to compute $a_8S(\theta)$; these two tasks account for 9 32-bit multiplications. The main cost is the computation of $S^2(\theta)$. Recall that $A(\theta)$ is of degree (at most) 8 and so $S(\theta)$ is of degree (at most) 7 having a total of 8 coefficients. Karatsuba with 2-way split can be applied with recursion depth 3 to compute $S^2(\theta)$ using a total of 27 32-bit multiplications. (We say more about this below.) So, $A^2(x)$ given by (23) can be computed using a total of 36 32-bit multiplications. As a result, for squaring there is no difference in the number of required multiplications using 3-way Karatsuba or hybrid Karatsuba.

There are two ways to consider squaring using 2-way Karatsuba. Suppose $\alpha(x) = \alpha_0 + x\alpha_1$ is to be squared. This can be obtained as either

$$\alpha^2(x) = \alpha_0^2 + 2\alpha_0\alpha_1 x + \alpha_1^2 x^2 \tag{24}$$

requiring 2 squarings and 1 multiplication; or, as

$$\alpha^2(x) = \alpha_0^2 + ((\alpha_0 + \alpha_1)^2 - \alpha_0^2 - \alpha_1^2)x + \alpha_1^2 x^2 \tag{25}$$

requiring 3 squarings. The second method performs a squaring using squarings and addition/subtraction of smaller objects, while the first method performs a squaring using squarings and multiplications of smaller objects.

While computing $S^2(\theta)$ using 2-way Karatsuba, there will be 3 levels of recursion, corresponding to $x = \theta^4$, $x = \theta^2$ and $x = \theta$. At the bottom level where $x = \theta$, the coefficients α_0 and α_1 are integers which are to be multiplied using 32-bit multiplication. Since there is no difference in time between 32-bit multiplication and 32-bit squaring, the method given by (24) is used, since this avoids the extra addition and subtraction required in the method given by (25). On the other hand, for the higher two levels corresponding to $x = \theta^4$ and $x = \theta^2$, the method given by (25) turns out to be faster. So, our implementation uses (25) at the higher two levels to break down the recursion and then at the bottom level, (24) is used to actually perform the integer multiplications. The results are then combined back up the recursion tree according to (25).

Given a proper polynomial $A(\theta)$ by $\mathcal{S}(A(\theta))$ we denote the polynomial $D(\theta)$ obtained as $D(\theta) = \mathcal{M}(A(\theta), A(\theta))$. The particular method of computing $\mathcal{S}(A(\theta))$ does not affect the result, though, it does affect the efficiency.

5.8 Hadamard Transform

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

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

5.9 Field Inversion

It is possible to compute the multiplicative inverse of a field element using exponentiation, i.e., for $r \in \mathbb{F}_p$, $r^{-1} = r^{2^{251}-11} \pmod{2^{251} - 9}$. The Montgomery inverse algorithm [36] is a faster alternative of the exponentiation method. Since p is a 251-bit integer, for any $r \in \{0, \ldots, p-1\}$, the Montgomery inverse algorithm on inputs r and p returns $s = r^{-1}2^{251} \bmod p$. The actual value of r^{-1} is obtained by computing $s9^{-1} \bmod p$. This is because $2^{251} \equiv 9 \bmod p$. It is also possible to implement Montgomery inversion in constant time using the algorithm described in [8].

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. For a proper polynomial $A(\theta)$, by $\mathcal{I}(A(\theta))$ we denote the inverse of $A(\theta)$.

5.10 Costs of Field Operations

Let $\mathbf{M}$ denote the cost of a single 32-bit multiplication and $\mathbf{A}$ denote the cost of a single 64-bit addition or subtraction. The costs of field operations are expressed in terms of $\mathbf{M}$ and $\mathbf{A}$. These costs are as follows.

- Field multiplication $\mathcal{M}$ or squaring $\mathcal{S}$: $44\mathbf{M} + 117\mathbf{A}$ using the hybrid Karatsuba method.
- Field squaring $\mathcal{S}$: $36\mathbf{M} + 74\mathbf{A}$ using the hybrid Karatsuba method.
- Field multiplication by a small constant $\mathcal{C}$: $9\mathbf{M} + 10\mathbf{A}$.
- Hadamard transform $\mathcal{C}$: $18\mathbf{A}$.

5.11 Choice of the Field

In Section 4.2, we have explained our choice of the particular prime $p = 2^{251} - 9$ and the advantage of this prime for SIMD instructions over the prime $2^{255} - 19$.

On the other hand, one may consider the possibility of using a composite order field with a large characteristic. One such choice is the field $\mathbb{F}_{(2^{127}-1)^2}$. This field has been used in [16] and the field $\mathbb{F}_{2^{127}-1}$ has been used in [4]. An element of $\mathbb{F}_{(2^{127}-1)^2}$ will be represented using two elements of $\mathbb{F}_{2^{127}-1}$. For overflow free arithmetic, each element of $\mathbb{F}_{2^{127}-1}$ will require a 5-limb representation. Using Karatsuba, a field multiplication in $\mathbb{F}_{(2^{127}-1)^2}$ will require 3 multiplications in $\mathbb{F}_{2^{127}-1}$; using the 5-limb representation of elements of $\mathbb{F}_{2^{127}-1}$, each multiplication in $\mathbb{F}_{2^{127}-1}$ will require 5 32-bit multiplications. So, a single multiplication in $\mathbb{F}_{(2^{127}-1)^2}$ will require 48 32-bit multiplications. Since this is larger than the number of 32-bit multiplications required for a single multiplication in $\mathbb{F}_{2^{251}-9}$, we did not explore $\mathbb{F}_{(2^{127}-1)^2}$ further.

6 Vector Operations

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 $A_k(\theta) = \sum_{i=0}^{8} a_{k,i}\theta^i$ 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 9×4 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 $\mathbf{A}(\theta) = \sum_{i=0}^{8} \mathbf{a}_i\theta^i$. 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 $a_0, \ldots, a_3$.
- $\mathsf{unpack}(\mathbf{a})$: 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 $a_0, \ldots, a_3$.
- $\mathsf{pack}(A_0(\theta), A_1(\theta), A_2(\theta), A_3(\theta))$: returns $\mathbf{A}(\theta)$ represented as $\mathbf{A}(\theta) = \sum_{i=0}^{8} \mathbf{a}_i\theta^i$, 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 $\mathbf{A}(\theta) = \sum_{i=0}^{8} \mathbf{a}_i\theta^i$, for $j = 0, 1, 2, 3$, $A_j(\theta) = \sum_{i=0}^{8} a_{j,i}\theta^i$ 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 $(B_0(\theta), B_1(\theta), B_2(\theta), B_3(\theta),)$.

- $\mathcal{M}^4(\mathbf{A}(\theta), \mathbf{B}(\theta))$: returns $\mathbf{C}(\theta) = \sum_{i=0}^{8} \mathbf{c}_i\theta^i$ 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 $k = 0, \ldots, 3$.
- $\mathcal{S}^4(\mathbf{A}(\theta))$: returns $\mathbf{C}(\theta) = \sum_{i=0}^{8} \mathbf{c}_i\theta^i$ representing $(C_0(\theta), C_1(\theta), C_2(\theta), C_3(\theta))$ where $C_k(\theta) = \mathsf{sqr}(A_k(\theta))$ for $k = 0, \ldots, 3$.
- $\mathcal{C}^4(\mathbf{A}(\theta), \mathbf{d})$: returns $\mathbf{C}(\theta) = \sum_{i=0}^{8} \mathbf{c}_i\theta^i$ 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 $k = 0, \ldots, 3$. Here each d_k is a small constant represented as a 64-bit word so that $\mathbf{d}$ is a 256-bit quantity.

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 $(a_0 b_0, a_1 b_1, a_2 b_2, a_3 b_3)$ with each component reduced modulo 2^{64}.

Using the above SIMD operations to replace the 32-bit operations in the algorithm for multiplying a pair of field elements directly provides an algorithm for multiplying four pairs of field elements. Similar vectorisation is achieved for squaring.

6.1 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 $\mathsf{dup}_2(\mathbf{a}) = (a_1, a_1, a_3, a_3)$.

$\mathcal{H}^2(\mathbf{A}(\theta))$
input: $\mathbf{A}(\theta) = \sum_{i=0}^{8} \mathbf{a}_i \theta^i$ representing $(A_0(\theta), A_1(\theta), A_2(\theta), A_3(\theta))$;
output: $\mathbf{C}(\theta) = \sum_{i=0}^{8} \mathbf{c}_i \theta^i$ 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 $i = 0, \ldots, 8$ do
2. $\quad \mathbf{s} = \mathsf{dup}_1(\mathbf{a}_i)$;
3. $\quad \mathbf{t} = \mathsf{dup}_2(\mathbf{a}_i)$;
4. $\quad \mathbf{t} = \mathbf{t} \oplus (0^{64}, 1^{64}, 0^{64}, 1^{64})$;
5. $\quad \mathbf{t} = \mathbf{t} + (0^{64}, 2\mathfrak{p}_i + 1, 0^{64}, 2\mathfrak{p}_i + 1)$;
6. $\quad \mathbf{c}_i = \mathbf{t} + \mathbf{s}$;
7. end for;

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

The goal is to compute $(A_0(\theta) + A_1(\theta), A_0(\theta) - A_1(\theta), A_2(\theta) + A_3(\theta), A_2(\theta) - A_3(\theta))$. Instead at the end of Step 7, $(A_0(\theta) + A_1(\theta), A_0(\theta) + 2\mathfrak{P}(\theta) - A_1(\theta), A_2(\theta) + A_3(\theta), A_2(\theta) + 2\mathfrak{P}(\theta) - A_3(\theta))$ is computed. The reduce operation ensures that the correct result is returned. For further explanation of how this is achieved, we refer to Section 5.4. The $\oplus$ operation is implemented using `_mm256_xor_si256`; the additions in Steps 5 and 6 can be implemented using `_mm256_add_epi32`; the operations dup_1 and dup_2 are implemented using `_mm256_permute4x64_epi64`.

6.2 Vector Duplication

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

Let $\mathbf{A}(\theta) = \sum_{i=0}^{8} \mathbf{a}_i \theta^i$ be a proper vector and b be a bit. We define the operation $\mathcal{P}^4(\mathbf{A}, b)$ to return $\sum_{i=0}^{8} \mathsf{copy}(\mathbf{a}_i, b)\theta^i$. If $\mathbf{A}(\theta)$ represents $(A_0(\theta), A_1(\theta), A_2(\theta), A_3(\theta))$, then

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

6.3 Costs of Vector Operations

The costs of vector operations are expressed in terms of the number of SIMD operations that are required. Let $\mathbf{A}^4$ denote the cost of one `_mm256_add_epi64` or one `_mm256_sub_epi64` operation; and let $\mathbf{M}^4$ denote the cost of one `_mm256_mul_epu32` operation. The costs of the various vector operations are as follows.

- Vectorised multiplication $\mathcal{M}^4$: $44\mathbf{M}^4 + 117\mathbf{A}^4$.
- Vectorised squaring $\mathcal{S}^4$: $36\mathbf{M}^4 + 74\mathbf{A}^4$.
- Vectorised multiplication by a constant $\mathcal{C}^4$: $9\mathbf{M}^4 + 10\mathbf{A}^4$.
- Vectorised Hadamard transform $\mathcal{H}^2$: $18\mathbf{A}^4$ along with 9 `_mm256_xor_si256` and 9 `_mm256_and_si256` instructions.
- $\mathcal{P}^4$: 9 `_mm256_permutevar8x32_epi32` instructions.

7 Vectorised Scalar Multiplication

Scalar multiplication on the Kummer line is computed from a base point $[x : z]$ 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 $\mathbb{F}_p$. If x^2 and z^2 are small as in the fixed base point of $\mathcal{K}_{101,61}$, then these two values are represented as 32-bit words (or even as bytes). In general, the values x^2 and z^2 will be arbitrary elements of $\mathbb{F}_p$ and will have a 9-limb representation as has been described above.

The scalar multiplication algorithm using vector operations is given below. In the algorithm below, a^2 and b^2 are the parameters of $\mathcal{K}_{a^2,b^2}$ while A^2 and B^2 are defined from a^2 and b^2 as $A^2 = (a^2 + b^2)/2$ and $B^2 = (a^2 - b^2)/2$.

$\mathsf{scalarMult}(P, n)$
Input: base point $P = (x^2, z^2)$ and ℓ-bit scalar n given as $(1, n_{\ell-2}, \ldots, n_0)$;
Output: $nP = (U^2(\theta), V^2(\theta))$;

1. $\mathfrak{a} = \mathsf{pack}(B^2, A^2, B^2, A^2)$;
2. $\mathfrak{c}_0 = \mathsf{pack}(b^2, a^2, z^2, x^2)$; $\mathfrak{c}_1 = \mathsf{pack}(b^2, a^2, z^2, x^2)$;
3. compute $2P = (X_2^2(\theta), Z_2^2(\theta))$;
4. $\mathbf{T}(\theta) = \mathsf{pack}(X^2, Z^2, X_2^2(\theta), Z_2^2(\theta))$;
5. for $i = \ell - 2$ down to 0
6. $\quad \mathbf{T}(\theta) = \mathcal{H}^2(\mathbf{T}(\theta))$;
7. $\quad \mathbf{S}(\theta) = \mathcal{P}^4(\mathbf{T}(\theta), n_i)$;
8. $\quad \mathbf{T}(\theta) = \mathcal{M}^4(\mathbf{T}(\theta), \mathbf{S}(\theta))$;
9. $\quad \mathbf{T}(\theta) = \mathcal{C}^4(\mathbf{T}(\theta), \mathfrak{a})$;
10. $\quad \mathbf{T}(\theta) = \mathcal{H}^2(\mathbf{T}(\theta))$;
11. $\quad \mathbf{T}(\theta) = \mathcal{S}^4(\mathbf{T}(\theta))$;
12. $\quad \mathbf{T}(\theta) = \mathcal{C}^4(\mathbf{T}(\theta), \mathfrak{c}_{n_i})$;
13. end for;
14. $(U^2(\theta), V^2(\theta), \cdot, \cdot) = \mathsf{unpack}(\mathbf{T}(\theta))$;
15. return $(U^2(\theta), V^2(\theta))$.

Remark: In the above description, we have assumed that the base point (x^2, z^2) is represented by small integers. This is true if the scalar multiplication is for a fixed base point. On the other hand, for variable base point, this is no longer true. In this case, in Step 12 it is required to use the operation $\mathcal{M}^4$ instead of the operation $\mathcal{C}^4$. In any case, since the base point is the same for each ladder step, all the ladder steps take the same time.

Correctness: Steps 6 to 12 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 $kP = (x_1^2(\theta), z_1^2(\theta))$ and $(k+1)P = (x_2^2(\theta), z_2^2(\theta))$. These two points are represented in packed form as $(x_1^2(\theta), z_1^2(\theta), x_2^2(\theta), z_2^2(\theta))$ which is the vector input to the ladder step. Denoting the output of the ladder step as $(x_3^2(\theta), z_3^2(\theta), x_4^2(\theta), z_4^2(\theta))$, 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.

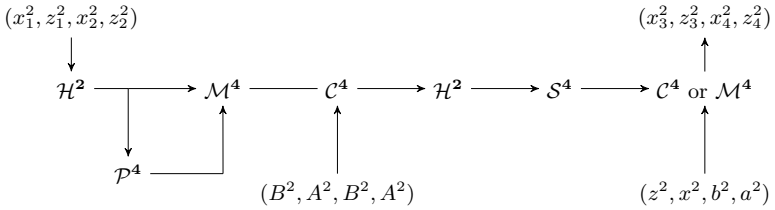

Fig. 4. One vectorized ladder step

Cost: In Step 2, $2P$ is computed which needs $2\mathcal{M} + 6\mathcal{S} + 8\mathcal{C} + 2\mathcal{H}$ field operations and so a total of $304\mathbf{M} + 678\mathbf{A}$ operations. Each vectorised ladder step requires $1\mathcal{M}^4 + 1\mathcal{S}^4 + 2\mathcal{C}^4 + 2\mathcal{H}^2 + 1\mathcal{P}^4$ and so a total of $98\mathbf{M}^4 + 247\mathbf{A}^4$ operations; plus 18 `_mm256_xor_si256` instructions; 18 `_mm256_and_si256` instructions; and 9 `_mm256_permutevar8x32_epi32` instructions.

Vectorised scalar multiplication with a 256-bit scalar requires $304\mathbf{M} + 678\mathbf{A} + 256(98\mathbf{M}^4 + 247\mathbf{A}^4)$ operations; plus 4608 `_mm256_xor_si256` instructions; 4608 `_mm256_and_si256` instructions; and 2304 `_mm256_permutevar8x32_epi32` instructions. Additionally, there is a cost of packing the data before starting the ladder steps and the cost of unpacking the data after all the ladder steps are over. Depending on the application, it may be required to convert the final Kummer point from projective to affine representation. The cost of doing this is $1\mathcal{M} + 1\mathcal{I}$.

8 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].

The experiments for $\mathcal{K}_{101,61}$ were performed on a machine with Intel®Core™i7-4790 Haswell 4-core CPU having four cores with each core running at 3.60GHz. The code runs on a single core. For further details of the Haswell architecture and AVX2 instructions, we refer to [24, 35]. The OS was 64-bit Ubuntu-14.04-LTS operating system and the C code was complied using GCC version 4.8.4. Timing measurements were performed using the methodology described at [31]. During measurement, turbo boost and hyperthreading were turned off. We used average from 100,000 iterations. Cache training was done using 25000 iterations. The Time Stamp Counter (TSC) was read from the CPU to RAX and RDX registers by RDTSC instruction. For the actual measurements, the header file "measurement.h", given in [31] was used.

Table 5 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 [45] and [32], the timings are obtained from the respective papers. For Curve25519, we downloaded the `sandy2x`[4] library and timed Curve25519 on the machine which was used to measure the time for $\mathcal{K}_{101,61}$ using the methodology from [31][5]. The cycle count of 158169 on Haswell that we obtain for Curve25519 is close to 156076 cycles reported by Tung Chou at `https://moderncrypto.org/mail-archive/curves/2015/000637.html` and the count of about 156500 cycles reported in [21]. Further, EBACS (`https://bench.cr.yp.to/results-dh.html`) also mentions about 156000 cycles on the machine `titan0`.

For fixed base scalar multiplication, $\mathcal{K}_{101,61}$ requires **106735** cycles while Curve25519 requires **144277** cycles. These measurements were made on the machine that was used for variable base scalar multiplication using the timing method from [31]. Again, for Curve25519, the `sandy2x` library was used.

Considering the comparison of Curve25519 and $\mathcal{K}_{101,61}$ we find that $\mathcal{K}_{101,61}$ gives significantly better performance for both variable base and fixed base scalar multiplications. We further note that our implementation of $\mathcal{K}_{101,61}$ is in Intel intrinsics, while the `sandy2x` library uses assembly (as does Four-$\mathbb{Q}$ and the implementations in [32, 45]). There is a possibility that an assembly implementation of $\mathcal{K}_{101,61}$ will show further competitive advantage over Curve25519.

It is possible to improve the speed of fixed base scalar multiplication using a pre-computed table along with a windowed-NAF scalar multiplication algorithm. Using this approach, [32] reports much faster timing for NIST P-256 and [14] reports much faster timing for Curve25519. We have not investigated the use of pre-computed tables for $\mathcal{K}_{101,61}$ and so we do not make a detailed comparison for this setting.

curve	genus	security	field	automorphism	cycles	pre-comp tab
Curve-25519 [2]	1	126	$\mathbb{F}_{2^{255}-19}$	no	**158169**	no
NIST P-256 [32]	1	128	$\mathbb{F}_{2^{256}-2^{224}+2^{192}+2^{96}-1}$	no	291000	no
Four-$\mathbb{Q}$ [16]	1	123	$\mathbb{F}_{(2^{127}-1)^2}$	yes	59000	2048 bits
				no	109000	no
$\mathcal{K}_{11,-22,-19,-3}$ [4]	2	125	$\mathbb{F}_{2^{127}-1}$	no	54389	no
Koblitz [45] (3-τNAF)	1	128	$\mathbb{F}_{4^{149}}$	yes	69,656	4768 bits
$\mathcal{K}_{101,61}$	1	124	$\mathbb{F}_{2^{251}-9}$	no	**141794**	no

Table 5. Timing comparison for variable base scalar multiplication.

9 Conclusion

This work has shown that compared to Curve25519, Kummer line based scalar multiplication for genus one 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. We do not claim to have the best possible implementation for $\mathcal{K}_{101,61}$. It may be possible to improve the speed by further optimisations. One possibility is to use assembly implementation to improve speed. Another possibility is to improve the speed of field multiplication by carefully implementing one of the other strategies which have been mentioned earlier and which require lesser number of 32-bit multiplications.

[4] Downloaded from `https://bench.cr.yp.to/supercop/supercop-20160910.tar.xz`. 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.

[5] We had initially intended to use SUPERCOP for timing $\mathcal{K}_{101,61}$ and had some email interactions with Peter Schwabe regarding this. An email from him on March 31, 2016 informed us that due to certain bugs in SUPERCOP some of the stated timing results in [4] cannot be reproduced. So, we opted instead to use [31] for timing both $\mathcal{K}_{101,61}$ and Curve25519.

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 on genus 2.

References

1. Diego F. Aranha, Paulo S. L. M. Barreto, Geovandro C. C. F. Pereira, and Jefferson E. Ricardini. A note on high-security general-purpose elliptic curves. Cryptology ePrint Archive, Report 2013/647, 2013. `http://eprint.iacr.org/2013/647`.
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 and Tanja Lange. Faster addition and doubling on elliptic curves. In Kaoru Kurosawa, editor, *Advances in Cryptology - ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer, 2007.
7. 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.
8. Joppe W. Bos. Constant time modular inversion. *J. Cryptographic Engineering*, 4(4):275–281, 2014.
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 [7], 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. Daniel R. L. Brown. SEC 2: Recommended elliptic curve domain parameters. `http://www.secg.org/sec2-v2.pdf`, 2010.
14. 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.
15. 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.
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. Curve25519. Wikipedia page on Curve25519. `https://en.wikipedia.org/wiki/Curve25519`, accessed 15th September, 2016.
19. Harold M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44:393–422, 2007.

20. 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.
21. 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.
22. E.V. Flynn. Formulas for Kummer on genus 2. `http://people.maths.ox.ac.uk/flynn/genus2/kummer/`, accessed on 15th September, 2016.
23. E.V. Flynn. The group law on the Jacobian of a curve of genus 2. *J. reine angew. Math.*, 439:45–69, 1993.
24. A. Fog. Software optimization resources. `http://agner.org/optimize/`, 2016.
25. Code for $\mathcal{K}_{101,61}$. `https://github.com/skarati/Kummer_Line`.
26. Steven D. Galbraith, Xibin Lin, and Michael Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 518–535. Springer, 2009.
27. Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200. Springer, 2001.
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. S. Gueron. Software optimizations for cryptographic primitives on general purpose x86_64 platforms. *Tutorial at IndoCrypt*, 2011.
32. Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptographic Engineering*, 5(2):141–151, 2015.
33. Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards curves revisited. In Josef Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings*, volume 5350 of *Lecture Notes in Computer Science*, pages 326–343. Springer, 2008.
34. Jun ichi Igusa. *Theta functions*. Springer, 1972.
35. Intel. Intrinsics guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#`.
36. Burton S. Kaliski Jr. The Montgomery inverse and its applications. *IEEE Trans. Computers*, 44(8):1064–1065, 1995.
37. Neal Koblitz. Elliptic curve cryptosystems. *Math. Comp.*, 48(177):203–209, 1987.
38. Neal Koblitz. Hyperelliptic cryptosystems. *J. Cryptology*, 1(3):139–150, 1989.
39. 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.
40. 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.
41. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.
42. D. Mumford. *Tata lectures on theta I*. Progress in Mathematics 28. Birkh äuser, 1983.
43. 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.
44. Thomaz Oliveira, Julio López, Diego F. Aranha, and Francisco Rodríguez-Henríquez. Lambda coordinates for binary elliptic curves. In Bertoni and Coron [7], pages 311–330.
45. 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.
46. Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 403–428. Springer, 2016.
47. NUMS: Nothing up my sleeve. `https://tools.ietf.org/html/draft-black-tls-numscurves-00`.