Noname manuscript No.
(will be inserted by the editor)

Two is the fastest prime: lambda coordinates for binary elliptic curves

Thomaz Oliveira · Julio López · Diego F. Aranha · Francisco Rodríguez-Henríquez

the date of receipt and acceptance should be inserted later

Abstract In this work, we present new arithmetic formulas for a projective version of the affine point representation $(x, x + y/x)$, for $x \neq 0$, which leads to an efficient computation of the scalar multiplication operation over binary elliptic curves. A software implementation of our formulas applied to a binary Galbraith-Lin-Scott elliptic curve defined over the field $\mathbb{F}_{2^{254}}$ allows us to achieve speed records for protected/unprotected single/multi-core random-point elliptic curve scalar multiplication at the 127-bit security level. When executed on a Sandy Bridge 3.4GHz Intel Xeon processor, our software is able to compute a single/multi-core unprotected scalar multiplication in $69,500$ and $47,900$ clock cycles, respectively; and a protected single-core scalar multiplication in $114,800$ cycles. These numbers are improved by around 2% and 46% on the newer Ivy Bridge and Haswell platforms, respectively, achieving in the latter a protected random-point scalar multiplication in 60,000 clock cycles.

1 Introduction

The Weierstrass form of an ordinary binary elliptic curve defined over $\mathbb{F}_q, q = 2^m$, is given by the equation

$$E/\mathbb{F}_q : y^2 + xy = x^3 + ax^2 + b, \tag{1}$$

with $a, b \in \mathbb{F}_q$ and $b \neq 0$. The set of affine points $P = (x, y)$ with $x, y \in \mathbb{F}_q$ that satisfy the above equation, together with the point at infinity $\mathcal{O}$, form an additive abelian group with respect to the elliptic point addition operation. This group is denoted as $E_{a,b}(\mathbb{F}_q)$. The number of points on the curve is denoted as $\#E_{a,b}(\mathbb{F}_q)$, and the integer $t = q + 1 - \#E_{a,b}(\mathbb{F}_q)$, known as the trace of Frobenius, satisfies $|t| \leq 2\sqrt{q}$.

Alternative affine representations for binary elliptic points, namely, $(x, \frac{y}{x})$ and $(x, x+\frac{y}{x})$, were introduced in [32,43]. In [43] this last representation was designated λ-affine representation of points, and was used for performing the point doubling operation in [38,39,43], for point halving in [32,44,17,6], and for point compression in [40].

From the algorithmic point of view, point representation is one of the most important factors to consider for obtaining an efficient scalar multiplication computation. Due to the relatively expensive cost of field multiplicative inversions associated with the arithmetic of the affine point representation, projective coordinate systems were introduced in the early nineties.

In the case of binary curves, one of the first proposals was the homogeneous projective coordinates system [1], which represents an affine point $P = (x, y)$ as the triplet (X, Y, Z), where $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$; whereas in the Jacobian coordinate system [13], a projective point (X, Y, Z) corresponds to the affine point $(x = \frac{X}{Z^2}, y = \frac{Y}{Z^3})$. In 1998, López-Dahab (LD) coordinates [38] were proposed using $x = \frac{X}{Z}$ and $y = \frac{Y}{Z^2}$. Since then, LD coordinates have become the most studied coordinate system for binary elliptic curves, with many authors [31,35,3,34,9] contributing to improve their performance. In 2007, Kim and Kim [30] presented a 4-dimensional LD coordinate system that represents P as (X, Y, Z, T^2), with $x = \frac{X}{Z}$, $y = \frac{Y}{T}$ and $T = Z^2$. In a different vein, Bernstein *et al.* introduced in [9] a set of complete formulas for binary Edwards elliptic curves.

The introduction in contemporary processor architectures of a native carry-less multiplier and vector instruction sets, such as Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX), has brought a renewed interest to the study of efficient and secure software implementations of scalar multiplication in elliptic curves defined over binary fields [46,5,4].

Among the works studying scalar multiplication over binary elliptic curves, the authors in [46] were the first

T. Oliveira and F. Rodríguez-Henríquez: A portion of this work was performed while the authors were visiting the University of Waterloo. The authors acknowledge partial support from the CONACyT project 132073.

J. López: The author was supported in part by the Intel Labs University Research Office.

T. Oliveira and F. Rodríguez-Henríquez
Computer Science Department, CINVESTAV-IPN, Mexico

J. López
Institute of Computing, University of Campinas, Brazil

D. F. Aranha
Department of Computer Science, University of Brasília, Brazil

to analyze the impact of using the carry-less multiplier in the computation of the scalar multiplication over Koblitz and NIST curves at the 112-, 128- and 192-bit security levels. This work also presented multi-core implementations of their algorithms based on the parallel formulations first given in [2]. The authors in [4] held the record for the fastest unprotected implementation of scalar multiplication at the 128-bit security level by employing the NIST-K283 curve, where a simple analogue of the 2-dimensional GLV (Gallant, Lambert and Vanstone) method was used. This record was recently broken in [36,37,15], where the authors merged the GLS (Galbraith, Lin and Scott) and GLV methods to achieve a 4-dimensional decomposition of scalars in a non-standardized twisted Edwards prime curve. For protected implementations, authors in [11] hold the speed records in a genus-2 hyperelliptic curve, and the authors in [15] held the record in genus-1 elliptic curves before the results reported in this paper.

Our contributions. This work further contributes to the advances in binary elliptic curve arithmetic by presenting a new projective coordinate system and its corresponding group law, which is based on the λ-affine representation of a point $P = (x, \lambda)$, where $\lambda = x + \frac{y}{x}$. The efficient group law associated to this coordinate system enables significant speedups in the main loop of the traditional double-and-add and halve-and-add scalar multiplication methods as well as the pre- and post-computation efforts associated to these approaches. The concrete application of λ-coordinates to the 2-dimensional GLS-GLV method combined with an efficient quadratic field arithmetic implementation allow us to claim the speed records at the 127-bit security level for single and multi-core unprotected scalar multiplication, improving by 24% and 21% the timings reported in [36,15], respectively. For protected single-core scalar multiplication, our timings improve by 49%, 17% and 4% the results reported in [46,37,11], respectively, while staying slower than the latest speed record by a 16% margin [15]. In the newer Haswell processor, the proposed implementations received a significant performance boost, which allows us to report further speed records for the unprotected and protected scenarios. [1]

2 Binary Field Arithmetic

A binary extension field $\mathbb{F}_{2^m}$ of order $q = 2^m$ can be constructed by taking an m-degree polynomial $f(x) \in \mathbb{F}_2[x]$ irreducible over $\mathbb{F}_2$. The $\mathbb{F}_{2^m}$ field is isomorphic to $\mathbb{F}_2[x]/(f(x))$ and its elements consist of the collection of binary polynomials of degree less than m. Quadratic extensions of a binary extension field can be built using a degree two monic polynomial $g(u) \in \mathbb{F}_2[u]$ that happens to be irreducible over $\mathbb{F}_q$. In this case, the field $\mathbb{F}_{q^2}$ is isomorphic to $\mathbb{F}_q[u]/(g(u))$ and its elements can be represented as $a + bu$, with $a, b \in \mathbb{F}_q$. In this paper, we developed an efficient field arithmetic library for the fields $\mathbb{F}_q$ and its quadratic extension $\mathbb{F}_{q^2}$, with $m = 127$, which were constructed by means of the irreducible trinomials $f(x) = x^{127} + x^{63} + 1$ and $g(u) = u^2 + u + 1$, respectively.

2.1 Field multiplication over $\mathbb{F}_q$

Given two field elements $a, b \in \mathbb{F}_q$, field multiplication can be performed by polynomial multiplication followed by modular reduction as, $c = a \cdot b \bmod f(x)$. Since the binary coefficients of the base field elements $\mathbb{F}_q$ can be packed as vectors of two 64-bit words, the usage of the standard Karatsuba method allows us to compute the polynomial multiplication step at a cost of three 64-bit products (equivalent to three invocations of the carry-less multiplication instruction [46]), plus some additions. Due to the very special form of $f(x)$, modular reduction is especially elegant as it can be accomplished using essentially additions and shifts (see Section 2.4).

2.2 Field squaring, square root and multi-squaring over $\mathbb{F}_q$

Due to the action of the Frobenius operator, field squaring and square-root are linear operations in any binary field [42]. These two operations can be implemented at a very low cost provided that the base field $\mathbb{F}_q$ is defined by a square-root friendly trinomial or pentanomial. Furthermore, vectorized implementations with simultaneous table lookups through byte shuffling instructions, as presented in [5], kept square and square-root efficient relative to multiplication even with the dramatic acceleration of field multiplication brought by the native carry-less multiplier.

Multi-squaring, or exponentiation to 2^k, with $k > 5$ is performed via look-up of per-field constant tables of field elements, as proposed in [2,10]. For fixed k, a table T of $2^4 \cdot \lceil \frac{m}{4} \rceil$ field elements can be precomputed such

[1] The benchmarking was run on Intel platforms Xeon E31270 3.4GHz (Sandy Bridge), Core i5 3570 3.4GHz (Ivy Bridge), and Core i7 4770K (Haswell). In addition, our library was submitted to the ECRYPT Benchmarking of Asymmetric Systems (eBATS) where it is publicly available.

that $T[j, i_0+2i_1+4i_2+8i_3] = (i_0z^{4j}+i_1z^{4j+1}+i_2z^{4j+2}+i_3z^{4j+3})^{2^k}$ and $a^{2^k} = \sum_{j=0}^{\lceil \frac{m}{4} \rceil} T[j, \lfloor a/2^{4j} \rfloor \bmod 2^4]$.

2.3 Field inversion over $\mathbb{F}_q$

Field inversion in the base field is carried out using the Itoh-Tsujii algorithm [27], by computing $a^{-1} = a^{(2^{m-1}-1)2}$. The exponentiation is computed through the terms $(a^{2^i-1})^{2^k} \cdot a^{2^k-1}$, with $0 \leq i \leq k \leq m-1$. The overall cost of the method is $m-1$ squarings and 9 multiplications given by the length of the following addition chain for $m-1 = 126$,

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 24 \rightarrow 48 \rightarrow 96 \rightarrow 120 \rightarrow 126.$$

The cost of squarings can be reduced by computing each required 2^k-power as a multi-squaring whenever $k > 5$.

2.4 Modular Reduction

Table 1 provides the notation of the vector instructions that were used for performing the modular reduction algorithms to be presented in this section. This notation is closely based on [5], but notice that here, we are invoking the three-operand AVX instructions corresponding to 128-bit SSE instructions. Bitwise logical instructions operate across two entire vector registers and produce the result in a third vector register. Bitwise shifts perform parallel shifts in the 64-bit integers packed in a vector register, not propagating bits between contiguous data objects and requiring additional instructions to implement 128-bit shifts. Bytewise shifts are different in both the shift amount, which must be a multiple of 8; and the propagation of shifted out bytes between the two operands. Byte interleaving instructions take bytes alternately from the lower or higher halves of two vector register operands to produce a third output register.

For our irreducible trinomial $f(x) = x^{127} + x^{63} + 1$ choice, we use the procedure shown in Algorithm 1, which requires ten vector instructions to perform a reduction in the base field $\mathbb{F}_q$. This modular reduction algorithm can be improved when performing field squaring. In this case, the 253-bit polynomial a^2, with $a \in \mathbb{F}_q$, is represented using two 128-bit registers $r_1||r_0$. By observing that the 63-th bit of the register r_1 is zero, the optimized modular reduction algorithm uses just six vector instructions, as shown in Algorithm 2.

Algorithm 1 Modular reduction by trinomial $f(x) = x^{127} + x^{63} + 1$.

```
Input: 253-bit polynomial $d'$ stored into two 128-bit registers $r_1||r_0$.
Output: $\mathbb{F}_q$ element $d'$ mod $f(x)$ stored into a 128-bit register $r_0$.
$t_0 \leftarrow (r_1, r_0) \rhd 64$
$t_0 \leftarrow t_0 \oplus r_1$
$r_1 \leftarrow r_1 \ll_{64} 1$
$r_0 \leftarrow r_0 \oplus r_1$
$r_1 \leftarrow inthi_{64}(r_1, t_0)$
$r_0 \leftarrow r_0 \oplus r_1$
$t_0 \leftarrow t_0 \gg_{64} 63$
$r_0 \leftarrow r_0 \oplus t_0$
$r_1 \leftarrow intlo_{64}(t_0, t_0)$
$r_0 \leftarrow r_0 \oplus (r_1 \ll_{64} 63)$
return $r_0$
```

Algorithm 2 Modular reduction by $f(x) = x^{127} + x^{63} + 1$ for the squaring operation.

```
Input: 253-bit polynomial $a^2$ stored into two 128-bit registers $r_1||r_0$.
Output: $\mathbb{F}_q$ element $a^2$ mod $f(x)$ stored into a 128-bit register $r_0$.
$t_0 \leftarrow (r_1, r_0) \rhd 64$
$t_0 \leftarrow t_0 \oplus r_1$
$r_1 \leftarrow r_1 \ll_{64} 1$
$r_0 \leftarrow r_0 \oplus r_1$
$t_0 \leftarrow inthi_{64}(r_1, t_0)$
$r_0 \leftarrow r_0 \oplus t_0$
return $r_0$
```

2.5 Half-trace over $\mathbb{F}_q$

The trace function on $\mathbb{F}_{2^m}$ is the function $Tr : \mathbb{F}_{2^m} \mapsto \mathbb{F}_2$ defined as $Tr(c) = \sum_{i=0}^{m-1} c^{2^i}$. The solutions of quadratic equations $x^2 + x = c$ over $\mathbb{F}_q$, with $Tr(c) = 0$, can be found by means of the half-trace function $H : \mathbb{F}_{2^m} \mapsto \mathbb{F}_{2^m}$, which is defined as $H(c) = \sum_{i=0}^{(m-1)/2} c^{2^{2i}}$. A fast computation of this function can be achieved by exploiting its linear property,

$$H(c) = H(\sum_{i=0}^{m-1} c_i x^i) = \sum_{i=0}^{m-1} c_i H(x^i),$$

and by using an 8-bit index look-up table T of size $2^8 \cdot \lceil \frac{m}{8} \rceil$ elements such that,

$$H(c) = \sum_{j=0}^{\lceil \frac{m}{8} \rceil} T[j, \lfloor \frac{a}{2^{8j}} \rfloor \bmod 2^8].$$

2.6 Field arithmetic over $\mathbb{F}_{q^2}$

Recall that the quadratic extension $\mathbb{F}_{q^2}$ of the base field $\mathbb{F}_q$ is built using the monic trinomial $g(u) = u^2 + u + 1 \in \mathbb{F}_2[u]$ irreducible over $\mathbb{F}_q$. An arbitrary field element $a \in \mathbb{F}_{q^2}$ is represented as $a = a_0 + a_1 u$, with $a_0, a_1 \in \mathbb{F}_q$. Operations in the quadratic extension are performed coefficient-wise. For instance, the multiplication of two elements $a, b \in \mathbb{F}_{q^2}$ is computed as,

$$\begin{aligned} a \cdot b &= (a_0 + a_1 u) \cdot (b_0 + b_1 u) \\ &= (a_0 b_0 + a_1 b_1) + (a_0 b_0 + (a_0 + a_1) \cdot (b_0 + b_1))u, \end{aligned}$$

with $a_0, a_1, b_0, b_1 \in \mathbb{F}_q$.

Table 1 Vector instructions used for the binary field arithmetic implementation.

Symbol	Description	AVX
$\oplus$, $\wedge$, $\vee$	Bitwise XOR, AND, OR	VPXOR, VPAND, VPOR
$\ll_{64}, \gg_{64}$	Bitwise shift of packed 64-bit integers	VPSLLQ, VPSRLQ
$\rhd$	Bytewise multi-precision shift	VPALIGNR
$intlo_{64}$, $intlhi_{64}$	Byte interleaving of packed 64-bit integers	VPUNPCKLBW, VPUNPCKHBW

The square and square-root of a field element a is accomplished using the identities,

$$a^2 = (a_0 + a_1 u)^2 = a_0^2 + a_1^2 + a_1^2 u,$$
$$\sqrt{a} = \sqrt{(a_0 + a_1 u)} = \sqrt{a_0 + a_1} + \sqrt{a_1} u,$$

respectively. The multiplicative inverse c of a field element a is found by solving the equation $a \cdot c = (a_0 + a_1 u)(c_0 + c_1 u) = 1$, which yields the unique solution, $c_0 = (a_0 + a_1)t^{-1}$ and $c_1 = a_1 t^{-1}$, where $t = a_0 a_1 + {a_0}^2 + {a_1}^2$.

Solving quadratic equations over $\mathbb{F}_{q^2}$ of the form $x^2 + x = c$ with $Tr(c) = 0$,[1] reduces to the solution of two quadratic equations over $\mathbb{F}_q$, as discussed next. For an element $a = a_0 + a_1 u \in \mathbb{F}_{q^2}$, a solution $x = x_0 + x_1 u \in \mathbb{F}_{q^2}$ to the quadratic equation $x^2 + x = a$, can be found by solving the base field quadratic equations,

$$\begin{aligned} x_0^2 + x_1^2 + x_0 &= a_0 \\ x_1^2 + x_1 &= a_1 \end{aligned}$$

Notice that, since $Tr(a_1) = 0$, the solution to the second equation above can be found as $x_1 = H(a_1)$. Then x_0 is determined from $x_0^2 + x_0 = x_1 + a_1 + a_0 + Tr(x_1 + a_1 + a_0)$. The solution is $x = x_0 + (x_1 + Tr(x_1 + a_1 + a_0))\, u$ [22].

The costs of the quadratic extension arithmetic in terms of its base field operations and C language implementation, are presented in Table 2. Throughout this paper, we denote by $(a_b, m_b, q_b,\ s_b, i_b, h_b, t_b)$ and $(\tilde{a}, \tilde{m}, \tilde{q}, \tilde{s}, \tilde{i}, \tilde{h}, \tilde{t})$ the computational effort associated to the addition, multiplication, square-root, squaring, inversion, half-trace and trace operations over $\mathbb{F}_q$ and $\mathbb{F}_{q^2}$, respectively.

3 Lambda projective coordinates

In order to have a more efficient elliptic curve arithmetic, it is standard to use a projective version of the Weierstrass elliptic curve equation (1), where the points are represented in the so-called projective space. In the following we describe the *λ-projective coordinates*, a coordinate system whose associated group law is introduced here for the first time.

Given a point $P = (x, y) \in E_{a,b}(\mathbb{F}_q)$ with $x \neq 0$, the λ-affine representation of P is defined as (x, λ), where $\lambda = x + \frac{y}{x}$. The λ-projective point $P = (X, L, Z)$ corresponds to the λ-affine point $(\frac{X}{Z}, \frac{L}{Z})$. The λ-projective equation form of the Weierstrass Equation (1) is,

$$(L^2 + LZ + aZ^2)X^2 = X^4 + bZ^4. \tag{2}$$

Notice that the condition x = 0 does not pose a limitation in practice, since the only point with x = 0 that satisfies Eq. (1) is $(0, \sqrt{b})$.

3.1 Group law

In this subsection, the formulas for point doubling and addition in the λ-projective coordinate system are presented. Complementary formulas and complete proofs of all theorems can be found in Appendix A.

Theorem 1. *Let $P = (X_P, L_P, Z_P)$ be a point in a non-supersingular curve $E_{a,b}(\mathbb{F}_q)$. Then the formula for $2P = (X_{2P}, L_{2P}, Z_{2P})$ using the λ-projective representation is given by*

$$\begin{aligned} T &= L_P^2 + (L_P \cdot Z_P) + a \cdot Z_P^2 \\ X_{2P} &= T^2 \\ Z_{2P} &= T \cdot Z_P^2 \\ L_{2P} &= (X_P \cdot Z_P)^2 + X_{2P} + T \cdot (L_P \cdot Z_P) + Z_{2P}. \end{aligned}$$

For situations where the multiplication by the b-coefficient is fast, one can replace a full multiplication with a multiplication by the constant $a^2 + b$. We present below an alternative formula for calculating L_{2P}:

$$\begin{aligned} L_{2P} = {} & (L_P + X_P)^2 \cdot ((L_P + X_P)^2 + T + Z_P^2) \\ & + (a^2 + b) \cdot Z_P^4 + X_{2P} + (a + 1) \cdot Z_{2P}. \end{aligned}$$

Theorem 2. *Let $P = (X_P, L_P, Z_P)$ and $Q = (X_Q, L_Q, Z_Q)$ be points in $E_{a,b}(\mathbb{F}_q)$ with $P \neq \pm Q$. Then the*

[1] See §2.5 for a definition of the trace function $Tr(c)$.

Table 2 Cost of the field arithmetic $\mathbb{F}_{q^2} \cong \mathbb{F}_q[u]/(u^2+u+1)$ with respect to the base field $\mathbb{F}_q$ and its C-language implementation. See §2.6 for the definition of the arithmetic operations $(a_b, m_b, q_b, s_b, i_b, h_b, t_b)$ and $(\tilde{a}, \tilde{m}, \tilde{q}, \tilde{s}, \tilde{i}, \tilde{h}, \tilde{t})$. 'PCLMULQDQ', 'SSE instr.' and 'tbl lkup.' stand for carry-less multiplication, 128-bit SSE/AVX vector instruction and table look-up, respectively.

Arithmetic over $\mathbb{F}_{q^2}$	**Multiplication ($\tilde{m}$)**	**Square-Root ($\tilde{q}$)**	**Squaring ($\tilde{s}$)**	**Inversion ($\tilde{i}$)**	**Half-Trace ($\tilde{h}$)**
Cost in terms of $\mathbb{F}_q$ arithmetic operations	$3m_b + 4a_b$	$2q_b + a_b$	$2s_b + a_b$	$i_b + 3m_b + 3a_b$	$2h_b + t_b + 2a_b$
Number of instructions invoked	9 PCLMULQDQ + 62 AVX instr.	37 AVX instr.	33 AVX instr.	36 PCLMULQDQ + 386 AVX instr. + 160 tbl lkup.	19 AVX instr. + 32 tbl lkup.

addition $P+Q = (X_{P+Q}, L_{P+Q}, Z_{P+Q})$ can be computed by the formulas

$$\begin{aligned}
A &= L_P \cdot Z_Q + L_Q \cdot Z_P \\
B &= (X_P \cdot Z_Q + X_Q \cdot Z_P)^2 \\
X_{P+Q} &= A \cdot (X_P \cdot Z_Q) \cdot (X_Q \cdot Z_P) \cdot A \\
L_{P+Q} &= (A \cdot (X_Q \cdot Z_P) + B)^2 \\
&\quad + (A \cdot B \cdot Z_Q) \cdot (L_P + Z_P) \\
Z_{P+Q} &= (A \cdot B \cdot Z_Q) \cdot Z_P.
\end{aligned}$$

Furthermore, we derive an efficient formula for computing the operation $2Q+P$, with the points Q and P represented in λ-projective and λ-affine coordinates, respectively.

Theorem 3. *Let $P = (x_P, \lambda_P)$ and $Q = (X_Q, L_Q, Z_Q)$ be points in the curve $E_{a,b}(\mathbb{F}_q)$. Then the operation $2Q + P = (X_{2Q+P}, L_{2Q+P}, Z_{2Q+P})$ can be computed as follows:*

$$\begin{aligned}
T &= L_Q^2 + L_Q \cdot Z_Q + a \cdot Z_Q^2 \\
A &= X_Q^2 \cdot Z_Q^2 + T \cdot (L_Q^2 + (a + 1 + \lambda_P) \cdot Z_Q^2) \\
B &= (x_P \cdot Z_Q^2 + T)^2 \\
X_{2Q+P} &= (x_P \cdot Z_Q^2) \cdot A^2 \\
Z_{2Q+P} &= (A \cdot B \cdot Z_Q^2) \\
L_{2Q+P} &= T \cdot (A + B)^2 + (\lambda_P + 1) \cdot Z_{2Q+P}.
\end{aligned}$$

Table 3 summarizes the costs of the following point operations when using the λ-projective coordinate system in an elliptic curve defined over the quadratic field $E/\mathbb{F}_{q^2}$,

- **full addition**: $R = P + Q$, with P, Q represented in λ-projective coordinates,
- **mixed addition**: $R = P+Q$, with P represented in λ-affine coordinates and Q represented in λ-projective coordinates,
- **doubling**: $R = 2P$, with P represented in λ-projective coordinates and,
- ***atomic* doubling and addition**: $R = 2Q + P$ with P represented in λ-affine coordinates and Q represented in λ-projective coordinates.

where the terms $\tilde{m}_a$ and $\tilde{m}_b$ denote the field multiplication by the curve constants a and b, respectively. For comparison purposes, the costs of these operations when using the López-Dahab (LD) projective coordinate system [38] are also included.[2]

3.2 GLS curves

In 2001, Gallant, Lambert and Vanstone (GLV) [19] presented a technique that uses efficient computable endomorphisms available in certain classes of elliptic curves that allows significant speedups in the scalar multiplication computation. Later, Galbraith, Lin and Scott (GLS) [18] constructed efficient endomorphisms for a broader class of elliptic curves defined over $\mathbb{F}_{q^2}$, where q is a prime number, showing that the GLV technique also applies to these curves. Subsequently, Hankerson, Karabina and Menezes investigated in [22] the feasibility of implementing the GLS curves over $\mathbb{F}_{2^{2m}}$. In the following, we introduce the GLS curves over binary fields and their endomorphism. Our description closely follows the one given in [22].

Let $q = 2^m$ and let $E/\mathbb{F}_q : y^2 + xy = x^3 + ax^2 + b$, with $a, b \in \mathbb{F}_q$, be a binary elliptic curve. Also, pick a field element $a' \in \mathbb{F}_{q^2}$ such that $Tr(a') = 1$, where Tr is the trace function from $\mathbb{F}_{q^2}$ to $\mathbb{F}_2$ defined as, $Tr : c \mapsto \sum_{i=0}^{2m-1} c^{2^i}$. Given $\#E(\mathbb{F}_q) = q + 1 - t$, it follows that $\#E(\mathbb{F}_{q^2}) = (q+1)^2 - t^2$. Let us define

$$\tilde{E}/\mathbb{F}_{q^2} : y^2 + xy = x^3 + a'x^2 + b, \qquad (3)$$

[2] Notice that the *atomic* doubling and addition operation is exclusive of the λ-projective coordinate system. For the sake of a fair comparison, in the second row and fifth column of Table 3, the cost of performing a non-atomic point doubling plus a mixed point addition using LD coordinates is reported.

Table 3 A cost comparison of the elliptic curve arithmetic on $E/\mathbb{F}_{q^2}$ using López-Dahab (LD) coordinates vs. the λ-projective coordinate system. The costs are given in terms of field arithmetic operations over $\mathbb{F}_{q^2}$

Coordinate system	**Full addition**	**Mixed addition**	**Doubling**	**Doubling and mixed addition**
López-Dahab	$13\tilde{m} + 4\tilde{s}$	$8\tilde{m} + \tilde{m}_a + 5\tilde{s}$	$3\tilde{m} + \tilde{m}_a + \tilde{m}_b + 5\tilde{s}$	$11\tilde{m} + 2\tilde{m}_a + \tilde{m}_b + 10\tilde{s}$
Lambda	$11\tilde{m} + 2\tilde{s}$	$8\tilde{m} + 2\tilde{s}$	$4\tilde{m} + \tilde{m}_a + 4\tilde{s}$ / $3\tilde{m} + \tilde{m}_a + \tilde{m}_b + 4\tilde{s}$	$10\tilde{m} + \tilde{m}_a + 6\tilde{s}$

with $\#\tilde{E}_{a',b}(\mathbb{F}_{q^2}) = (q-1)^2 + t^2$. It is known that $\tilde{E}$ is the quadratic twist of E, which means that both curves are isomorphic over $\mathbb{F}_{q^4}$ under the endomorphism [22],

$$\phi : E \to \tilde{E}, (x, y) \mapsto (x, y + sx),$$

with $s \in \mathbb{F}_{q^4} \backslash \mathbb{F}_{q^2}$ satisfying $s^2 + s = a + a'$. It is also known that the map ϕ is an involution, *i.e.*, $\phi = \phi^{-1}$. Let $\pi : E \to E$ be the Frobenius map defined as $(x, y) \mapsto (x^{2^m}, y^{2^m})$, and let ψ be the composite endomorphism $\psi = \phi\pi\phi^{-1}$ given as,

$$\psi : \tilde{E} \to \tilde{E}, (x, y) \mapsto (x^{2^m}, y^{2^m} + s^{2^m} x^{2^m} + s x^{2^m}).$$

In this work, the binary elliptic curve $\tilde{E}_{a',b}(\mathbb{F}_{q^2})$ was defined with the parameters $a' = u$ and $b \in \mathbb{F}_q$, where b was carefully chosen to ensure that $\#\tilde{E}_{a',b}(\mathbb{F}_{q^2}) = hr$, with $h = 2$ and where r is a prime of size $2m - 1$ bits. Moreover, $s^{2^m} + s = u$, which implies that the endomorphism ψ acting over the λ-affine point

$$P = (x_0 + x_1 u, \lambda_0 + \lambda_1 u) \in \tilde{E}_{a',b}(\mathbb{F}_{q^2}),$$

can be computed with only three additions in $\mathbb{F}_q$ as,

$$\psi(P) \mapsto ((x_0 + x_1) + x_1 u, (\lambda_0 + \lambda_1) + (\lambda_1 + 1)u).$$

It is worth to remark that in order to prevent the generalized Gaudry-Hess-Smart (gGHS) attack [20,25], the constant b of $\tilde{E}_{a',b}(\mathbb{F}_{q^2})$ must be carefully verified. Above remark notwithstanding, the probability that a randomly selected $b \in \mathbb{F}_q$ is a weak parameter, is negligibly small [22].

4 Scalar Multiplication

Let $\langle P \rangle$ be an additively written subgroup of prime order r defined over $\tilde{E}_{a',b}(\mathbb{F}_{q^2})$ (see Equation (3)). Let k be a positive integer such that $k \in [0, r-1]$. Then, the scalar multiplication operation, denoted by $Q = kP$, corresponds to adding P to itself $k - 1$ times. The average cost of computing kP by a random n-bit scalar k using the customary double-and-add method is about $nD + \frac{n}{2}A$, where D is the cost of doubling a point (i.e. the operation of computing $R = 2S = S + S$, with $S \in \langle P \rangle$) and A is the cost of a point addition (i.e. the operation of computing $R = S + T$, with $S, T \in \langle P \rangle$). Given a subgroup $\langle P \rangle$ of prime order r and a point $Q \in \langle P \rangle$, the Elliptic Curve Discrete Logarithm Problem (ECDLP) consists of finding the unique integer $k \in [0, r-1]$ such that $Q = kP$ holds.

In this section, the most prominent methods for computing the scalar multiplication on Weierstrass binary curves are described. Here, we are specifically interested in the problem of computing the elliptic curve scalar multiplication $Q = kP$, where $P \in \tilde{E}_{a',b}(\mathbb{F}_{q^2})$ is a generator of prime order r and $k \in \mathbb{Z}_r$ is a scalar of bitlength $|k| \approx |r| = 2m - 1$.

4.1 The GLV method and the w-NAF representation

Let ψ be a nontrivial efficiently computable endomorphism of $\tilde{E}$. Also, let us define the integer $\delta \in [2, r-1]$ such that $\psi(Q) = \delta Q$, for all $Q \in \tilde{E}_{a',b}(\mathbb{F}_{q^2})$. Computing kP via the GLV method consists of the following steps.

First, a balanced length-two representation of the scalar $k \equiv k_1 + k_2\delta \mod r$, must be found, where $|k_1|$, $|k_2| \approx |r|/2$. Given k and δ, there exist several methods to find k_1, k_2 [23,41,29]. However, with the protected implementation as the only exception, we decided to follow the suggestion in [18] which selects two integers k_1, k_2 at random, perform the scalar multiplication and then return $k \equiv k_1 + k_2\delta \mod r$, if required.[1] Having split the scalar k into two parts, the computation of $kP = k_1 P + k_2 \psi(P)$ can be performed by simultaneous multiple point multiplication techniques [24], in combination with any of the methods to be described next. A further acceleration can be achieved by representing the scalars k_1, k_2 in the width-w non-adjacent form (w-NAF). In this representation, k_j is written as an n-bit string $k_j = \sum_{i=0}^{n-1} k_{j,i} 2^i$, with $k_{j,i} \in \{0, \pm 1, \pm 3, \ldots, \pm 2^{w-1} - 1\}$, for $j \in \{1, 2\}$. A w-NAF string has a length $n \leq |k_j| + 1$, at most one nonzero bit among any

[1] We stress that k can be recovered at a very low computational effort. From our experiments, the scalar k could be reconstructed with cost slower than $5\tilde{m}$.

w consecutive bits, and its average nonzero-bit density is approximately $1/(w+1)$.

4.2 Left-to-right double-and-add

The computation of the scalar multiplication $kP = k_1P + k_2\psi(P)$ via the traditional left-to-right double-and-add method, can be achieved by splitting the scalar k as described above and representing the scalars k_1, k_2 so obtained in their w-NAF form. The precomputation step is accomplished by calculating the 2^{w-2} multiples $P_i = iP$ for odd $i \in \{1, \ldots, 2^{w-1}-1\}$. For the sake of efficiency, the multiples must be computed in λ-projective form, a task that can be accomplished using the atomic doubling and addition operation described in §3.1. This is followed by the application of the endomorphism to each point P_i so that the multiples $\psi(P_i)$ are also precomputed and stored. The computational effort associated with the precomputation is $38\tilde{m} + 2\tilde{m}_a + 8\tilde{s} + \tilde{i}$. Thereafter, the accumulator Q is initialized at the point at infinity $\mathcal{O}$, and the digits $k_{j,i}$ are scanned from left to right, one at a time. The accumulator is doubled at each iteration of the main loop and in case that $k_{j,i} \neq 0$, the corresponding precomputed multiple is added to the accumulator as, $Q = Q \pm P_{k'_i}$. Algorithm 3, with $t = 0$ illustrates the method just described.

4.3 Right-to-left halve-and-add

In the halve-and-add method [32,45], all point doublings are replaced by an operation called point halving. Given a point P, the halving point operation finds R such that $P = 2R$. For the field arithmetic implementation considered in this paper, the halving operation is faster than point doubling when applied on binary curves with $Tr(a') = 1$. Halving a point involves computing a field multiplication, a square-root extraction and solving a quadratic equation of the form $x^2 + x = c$ [17], whose solution can be found by calculating the half-trace of the field element c, as discussed in Section 2.

The halve-and-add method is described as follows. First, let us compute $k' \equiv 2^{n-1}k \mod r$, with $n = |r|$. This implies that, $k \equiv \sum_{i=0}^{n-1} k'_{n-1-i}/2^i + 2k'_n \mod r$ and therefore,

$$kP = \sum_{i=0}^{n-1} k'_{n-1-i}(\frac{1}{2^i}P) + 2k'_nP.$$

Then, k' is represented in its w-NAF form, and 2^{w-2} accumulators are initialized as, $Q_i = \mathcal{O}$, for $i \in \{1, 3, \ldots, 2^{w-1}-1\}$. Thereafter, each one of the n bits of k' are scanned from right to left. Whenever a digit $k'_i \neq 0$, the point $\pm P$ is added to the accumulator $Q_{k'_i}$, followed by $P = \frac{1}{2}P$, otherwise, only the halving of P is performed. In a final post-processing step, all the accumulators are added as $Q = \sum iQ_i$, for $i \in \{1, 3, \ldots, 2^{w-1}-1\}$. This summation can be efficiently accomplished using Knuth's method [33].[1] The algorithm outputs the result as $Q = kP$. Algorithm 3, with $t = n$ shows a two-dimensional GLV halve-and-add method.

Table 4 presents the estimated costs of the scalar multiplication algorithms in terms of point doublings (D), halvings (H), additions (A), Doubling and additions (DA) and endomorphisms (ψ) when performing the scalar multiplication in the curve $\tilde{E}_{a',b}(\mathbb{F}_{q^2})$.

4.4 Lambda-coordinates aftermath

Besides enjoying a slightly cheaper, but at the same time noticeable, computational cost when compared to the LD coordinates, the flexibility of the λ-coordinate system can improve the customary scalar multiplication algorithms in other more subtle ways. For instance, in the case of the double-and-add method, the usage of the atomic doubling and addition operation saves multiplications whenever an addition must be performed in the main loop. The speedup comes from the difference between the cost of the atomic doubling and addition $(10\tilde{m} + \tilde{m}_a + 6\tilde{s})$ shown in Table 3 versus the expense of naively performing a doubling and then adding the points in two separate steps $(12\tilde{m} + \tilde{m}_a + 6\tilde{s})$. To see the overall impact of this saving in say, the GLV double-and-add method, one has to calculate the probabilities of one, two or no additions in a loop iteration (the details of this calculation can be found in Appendix B). As mentioned before, it is also possible to apply the doubling and addition operation to speedup the calculation of the multiples of P in the precomputation phase. For that, we modified the original doubling and addition operation to compute *simultaneously* the points, $R, S = 2Q \pm P$, with an associate cost of just $16\tilde{m} + \tilde{m}_a + 8\tilde{s}$.

More significantly, there is an important multiplication saving in each one of the point additions in the main loop of the halve-and-add method. This is because points in the λ-form $(x, x + \frac{y}{x})$, are already in the required format for the λ-mixed addition operation and therefore, do not need to be reconverted to the regular affine representation as done in [17].

[1] For $w = 4$, the method is described as follows. $Q_5 = Q_5 + Q_7$, $Q_3 = Q_3 + Q_5$, $Q_1 = Q_1 + Q_3$, $Q_7 = Q_7 + Q_5 + Q_3$, $Q = 2Q_7 + Q_1$, which requires six point additions and one point doubling.

Table 4 Operation counts for selected scalar multiplication methods in $\tilde{E}_{a',b}(\mathbb{F}_{q^2})$

		Double-and-add	Halve-and-add
No-GLV	pre/post	$1D + (2^{w-2} - 1)A$	$1D + (2^{w-1} - 2)A$
(LD)	sc. mult.	$\frac{n}{w+1}A + nD$	$\frac{n}{w+1}(A + \tilde{m}) + nH$
2-GLV	pre/post	$1D + (2^{w-2} - 1)A + 2^{w-2}\psi$	$1D + (2^{w-1} - 2)A$
(LD)	sc. mult.	$\frac{n}{w+1}A + \frac{n}{2}D$	$\frac{n}{w+1}(A + \tilde{m}) + \frac{n}{2}H + \frac{n}{2(w+1)}\psi$
2-GLV	pre/post	$1D + (2^{w-2} - 1)A + 2^{w-2}\psi$	$1D + (2^{w-1} - 2)A$
(λ)	sc. mult.	$\frac{(2w+1)n}{2(w+1)^2}DA + \frac{w^2 n}{2(w+1)^2}D + \frac{n}{2(w+1)^2}A$	$\frac{n}{w+1}A + \frac{n}{2}H + \frac{n}{2(w+1)}\psi$

The concrete gains obtained from the λ-projective coordinates can be better appreciated in terms of field operations. Specifically, using the 4-NAF representation of a 254-bit scalar yields the following estimated savings. The double-and-add strategy requires $872\tilde{m} + 889\tilde{s}$ (considering $\tilde{m}_b = \frac{2}{3}\tilde{m}$) and $823\tilde{m} + 610\tilde{s}$ when performed with LD and λ-coordinates, respectively. This amounts for a saving of 31% and 5% in the number of field squarings and multiplications, respectively. The halve-and-add requires $772\tilde{m} + 255\tilde{s}$ and $721\tilde{m} + 101\tilde{s}$ when using LD and λ-coordinates, respectively. The savings that the latter coordinate system yields for this case are 60% and 6% fewer field squarings and multiplications, respectively. Notice that these estimations do not consider pre/postcomputation costs.

4.5 Parallel scalar multiplication

In this Section, we apply the method given in [2] for computing a scalar multiplication using two processors. The main idea is to compute $k'' \equiv 2^t k \mod r$, with $0 < t \leq n$. This produces,

$$k \equiv k''_{n-1}2^{n-1-t} + \ldots + k''_t 2^0 + k''_{t-1}/2^{-1} + \ldots + k''_0 2^{-t} \bmod r,$$

which can be rewritten as,

$$kP = \sum_{i=t}^{n-1} k''_i (2^{i-t}P) + \sum_{i=0}^{t-1} k''_i \left(\frac{1}{2^{-(t-i)}} P \right).$$

This parallel formulation allows to compute $Q = kP$ using the double-and-add and halve-and-add concurrently, where a portion of k is processed in different cores. The optimal value for the constant t depends on the performance of the scalar multiplication methods and therefore must be found experimentally. The GLV method combined with the parallel technique just explained is presented in Algorithm 3.[3]

Algorithm 3 Parallel GLV scalar multiplication

```
Input: $P \in E(\mathbb{F}_{2^{2m}})$, scalars $k_1, k_2$ of bitlength $n \approx |r|/2$,
  width $w$, constant $t$
Output: $Q = kP$
                Calculate $w$-NAF$(k_i)$ for $i \in \{1, 2\}$

  for $i \in \{1, \ldots, 2^{w-1}-1\}$ do        for $i \in \{1, \ldots, 2^{w-1}-1\}$ do
      Compute $P_i = iP$ and                        Initialize $Q_i \leftarrow \mathcal{O}$
      $\tilde{P}_i = \psi(P_i)$
  Initialize $Q_0 \leftarrow \mathcal{O}$

  for $i = n$ downto $t$ do                       for $i = t-1$ downto $0$ do
    $Q_0 \leftarrow 2Q_0$                             if $k_{1,i} > 0$ then
    if $k_{1,i} > 0$ then                               $Q_{k_{1,i}} \leftarrow Q_{k_{1,i}} + P$
      $Q_0 \leftarrow Q_0 + P_{k_{1,i}}$                  if $k_{1,i} < 0$ then
    if $k_{1,i} < 0$ then                               $Q_{k_{1,i}} \leftarrow Q_{k_{1,i}} - P$
      $Q_0 \leftarrow Q_0 - P_{k_{1,i}}$
                                                  if $k_{2,i} > 0$ then
    if $k_{2,i} > 0$ then                               $Q_{k_{2,i}} \leftarrow Q_{k_{2,i}} + \psi(P)$
      $Q_0 \leftarrow Q_0 + \tilde{P}_{k_{2,i}}$          if $k_{2,i} < 0$ then
    if $k_{2,i} < 0$ then                               $Q_{k_{2,i}} \leftarrow Q_{k_{2,i}} - \psi(P)$
      $Q_0 \leftarrow Q_0 - \tilde{P}_{k_{2,i}}$          $P \leftarrow P/2$
  end for                                       end for

                                                $Q \leftarrow \sum_{i \in \{1,\ldots,2^{w-1}-1\}} iQ_i$
  {Barrier}                                     {Barrier}
                Recode $k_1, k_2 \rightarrow k$, if necessary.
  return  $Q \leftarrow Q + Q_0$
```

4.6 Protected scalar multiplication

Regular scalar multiplication algorithms attempt to prevent leakage of information about the (possibly secret) scalar, obtained from procedures having non-constant execution times. There are two main approaches to make a scalar multiplication regular: one is using unified point doubling and addition formulas [9] and the other is recoding the scalar in a predictable pattern [28]. Both halve-and-add and double-and-add methods can be modified in the latter manner, with the additional care that table look-ups to read or write critical data need to be completed in constant-time. This can be accomplished by performing linear passes with conditional move instructions over the accumulators or precomputed points, thus thwarting cache-timing attacks.

[3] The pseudo-instruction *Barrier* refers to an OpenMP synchronization clause that forces each thread to wait until all the other threads have completed their assigned tasks.

Implementing timing-attack resistance usually impose significant performance penalties. For example, the density of regular recodings $(\frac{1}{w-1})$ is considerably lower than w-NAF and access to precomputed data becomes more expensive due to the linear passes. Efficiently computing a point halving in constant time is specially challenging, since the fastest methods for half-trace computation require considerable amounts of memory. This requirement can be relaxed if we assume that the base points are public information and available to the attacker. Notice however that this is a reasonable assumption in most protocols based on elliptic curves, but there are exceptions [12]. In any case, performing linear passes to read and store each one of the 2^{w-2} accumulators used in the halve-and-add procedure discussed in §4.3, impose a significant impact performance at every point addition.

Because of the above rationale, doubling-based methods seem to be a more promising option for protected implementations. Somewhat surprisingly, the regular recoding method combined with λ-coordinates admits an atomic formula for computing mixed addition plus doubling-and-addition as, $2Q + P_i + P_j$ with a cost of $17\tilde{m} + \tilde{m}_a + 8\tilde{s}$, saving one multiplication compared to performing the additions separately. Reading the points P_i, P_j can also be optimized by performing a single linear pass over the precomputed table. These optimizations alone are enough to compensate the performance gap between point doubling and point halving computations to be presented in the next section.

The approach for protected scalar multiplication is shown in Algorithm 4. In this procedure, the scalar k is decomposed into subscalars k_1, k_2 before the main loop. Because the regular recoding requires the input scalar to be odd, we modified slightly the GLV recoding algorithm to produce k_2 always odd, with at most one extra point addition needed to correct the result at the end. This is actually faster than generating random and possibly even k_1, k_2 for reconstructing k, because otherwise two point additions would be needed for correction. These extra point additions must always be performed for satisfying constant-time execution, but conditional move instructions can be used to eliminate incorrect results.

5 Results and discussion

Our library targets the Intel Sandy Bridge processor family. This multi-core micro-architecture supports carry-less multiplications, the SSE set of instructions [26] that operates on 128-bit registers and the AVX extension [16], which provides SIMD instructions in a three-operand format. However, our code can be easily adapted to any architecture that supports the aforementioned features. The benchmarking was run on an Intel Xeon E31270 3.4GHz and on an Intel Core i5 3570 3.4GHz with the TurboBoost and the HyperThreading technologies disabled. The code was implemented in the C programming language with intrinsics for vector instructions, compiled with GCC 4.8.1 and executed on 64-bit Linux. Experiments with the ICC 13.0 were also carried out and generated similar results. Portions of the code critical for timing-attack resistance (linear passes over precomputed tables, for example), were implemented in Assembly language to prevent undue manipulation by a code-optimizing compiler. For that reason, we abstain from presenting timings for that compiler. The main parameters of the GLS curve implemented in this work (elliptic curve constants, base point, order of the curve) are given in Appendix C.

Algorithm 4 Protected scalar multiplication

```
Input: $P \in E(\mathbb{F}_{2^{2m}})$, $k \in \mathbb{Z}$, width $w$
Output: $Q = kP$

  Decompose $k$ into $k_1, k_2$, with $k_2$ always odd.
  $c \leftarrow 1 - (k_1 \bmod 2)$
  $k_1 \leftarrow k_1 + c$
  Compute width-$w$ length-$l$ regular recodings of $k_1, k_2$.

  for $i \in \{1, \ldots, 2^{w-1} - 1\}$ do
      Compute $P_i = iP$

  $Q \leftarrow P_{k_{1,l-1}} + \psi(P_{k_{2,l-1}})$
  for $i = l - 2$ downto 0 do
      $Q \leftarrow 2^{w-2}Q$
      Perform a linear pass to recover $P_{k_{1,i}}, P_{k_{2,i}}$.
      $Q \leftarrow 2Q + P_{k_{1,i}} + \psi(P_{k_{2,i}})$
  end for

  return $Q \leftarrow Q - cP$
```

In the rest of this section, performance results for our software implementation of field arithmetic, elliptic point arithmetic and elliptic curve scalar multiplication are presented.

5.1 Field and elliptic curve arithmetic timings

Table 5 shows that the quadratic field arithmetic can handle the base field elements with a considerable efficiency. Field inversion, squaring and square-root, as well as the half-trace computational costs are just 1.27, 1.44, 1.87 and 1.43 times higher than their corresponding base field operations, respectively. Field multiplication in the quadratic field can be accomplished at a cost of about 2.23 times base field multiplications, which is significantly better than the theoretical Karatsuba ratio of three.

Table 5 Timings (in clock cycles) for the field arithmetic and elliptic curve operations in the Intel Sandy Brige platform.

Field operation	$\mathbb{F}_{2^{127}}$		$\mathbb{F}_{2^{254}}$	
	cycles	op/M[1]	cycles	op/M
Multiplication	42	1.00	94	1.00
Mod. Reduction[2]	6	0.14	11	0.12
Square root	8	0.19	15	0.16
Squaring	9	0.21	13	0.14
Multi-Squaring	55	1.31	n/a[3]	n/a
Inversion	765	18.21	969	10.30
Half-Trace	42	1.00	60	0.64
Trace	≈ 0	0	≈ 0	0

Elliptic curve operation	GLS $E/\mathbb{F}_{2^{254}}$	
	cycles	op/M
Doubling	450	4.79
Full addition	1102	11.72
Mixed addition	812	8.64
Doubling and add.	1063	11.30
Halving	233	2.48
No-GLV 4-NAF rec.	1540	16.38
2-GLV-4-NAF rec.	918	9.76
Reverse recoding	396	4.21

[1] Ratio to multiplication.

[2] This cost is included in the timings of all operations that require modular reduction.

[3] Multi-Squaring is used for the inversion algorithm, which is computed only in $\mathbb{F}_{2^{127}}$.

The lazy reduction technique was employed to optimize the λ-coordinate formulas. Nevertheless, experimental results showed us that this method should be used with caution. Extra savings were obtained by considering the separate case of performing mixed addition where the two points have their Z coordinate equal to one. In this case, mixed addition can be performed with just five multiplications and two squarings. This observation helped us to save more than 1000 cycles in the halve-and-add algorithm computation. The reverse recoding calculation, that is, given k_1, k_2 return $k \equiv k_1 + k_2\delta \mod r$ can be omitted if not required. However, in our scalar multiplication timings, this operation was included in all the cases.

5.2 Scalar multiplication timings

From both algorithmic analysis and experimental results considerations, we decided to use $w = 4$ for the w-NAF scalar recoding and $w = 5$ for the regular recoding from [28]. In the case of our parallel implementation (see Algorithm 3), the parameter $t = 72$ was selected, which is consistent with the 1.29 ratio between the double-and-add and halve-and-add computational costs. Notice that in the scalar multiplication procedure, it was assumed that the points are given and returned in the λ-affine form. If the input and output points must be represented in conventional affine coordinates, it is necessary to add about 1000 cycles ($2\tilde{m}+\tilde{i}$) to convert from conventional affine coordinates to the λ ones at the beginning and at the end of the scalar multiplication procedure. Furthermore, we observed an average 2% speedup when executing our code in the newer Ivy Bridge platform. Our scalar multiplication timings, along with the state-of-the-art implementations, are presented in Table 6.

5.2.1 Comparison to related work

Our single-core 4-NAF 2-dimensional GLV implementation achieves 69,500 clock cycles with the halve-and-add method. This result is 20% and 30% faster than the best implementations of point multiplication at the 128-bit security level over prime [15] and binary curves [4], respectively. Furthermore, our two-core parallel implementation using the GLV technique combined with the halve-and-add and double-and-add methods takes 47,900 clock cycles, thus outperforming by 21% the timings reported in [36] for a four-core parallel implementation. Also, the single and multi-core implementations at the 112-bit security level using Koblitz binary curves reported in [46] outperforms our code by just 2% and 3%, respectively. Finally, our single-core protected multiplication is 16% faster than [36], 4% faster than [11] and 16% slower than the current speed record on prime curves [15], but sets a new speed record for binary curves with an improvement of 49% compared to the previous one [46].

5.2.2 A field multiplication comparative

Trying to have a fair comparison that attenuates the diversity of curves, methods and technologies, Table 7 compares the estimated number of field multiplications required by implementations that represent the state-of-the-art of unprotected implementations of scalar multiplication computations.

The scalar multiplications on Koblitz curves reported in [46] and [4] require 13% and 20% less number of field multiplications than our work (2-GLV halve-and-add with λ-coordinates), respectively. However, since our field multiplication cost is 6% and 34% faster, our computational timings outperforms [4] and are competitive with [46], as seen in Table 6. This leads us to

Table 6 Timings (in clock cycles) for scalar multiplication with or without timing-attack resistance (TAR) in the Intel Sandy Bridge platform. In our implementation we always assume that the input and output points are provided in λ-affine coordinates.

Scalar multiplication	Curve	Security	Method	TAR	Cycles
Taverne *et al.* [46][2]	NIST-K233	112	No-GLV (τ-and-add)	no	67,800
Bos *et al.* [11][1]	BK/FKT	128	4-GLV (double-and-add)	no	156,000
Aranha *et al.* [4][2]	NIST-K283	128	2-GLV (τ-and-add)	no	99,200
Longa and Sica [36][2]	GLV-GLS	128	4-GLV (double-and-add)	no	91,000
Faz-H. *et al.* [15][2]	GLV-GLS	128	4-GLV, (double-and-add)	no	87,000
Taverne *et al.* [46][2]	NIST-K233	112	No-GLV, parallel (2 cores)	no	46,500
Longa and Sica [36][2]	GLV-GLS	128	4-GLV, parallel (4 cores)	no	61,000
Taverne *et al.* [46][2]	Curve2251	128	Montgomery ladder	yes	225,000
Bernstein [7,8][2]	Curve25519	128	Montgomery ladder	yes	194,000
Hamburg [21][3]	Montgomery	128	Montgomery ladder	yes	153,000
Longa and Sica [36][2]	GLV-GLS	128	4-GLV (double-and-add)	yes	137,000
Bos *et al.* [11][1]	Kummer	128	Montgomery ladder	yes	117,000
Faz-H. *et al.* [15][2]	GLV-GLS	128	4-GLV, (double-and-add)	yes	96,000
This work	GLS	127	2-GLV (double-and-add) (LD)	no	116,700
			2-GLV (double-and-add) (λ)	no	92,800
			2-GLV (halve-and-add) (LD)	no	82,800
			2-GLV (halve-and-add) (λ)	no	**69,500**
			2-GLV, parallel (2 cores) (λ)	no	**47,900**
			2-GLV (double-and-add) (λ)	yes	**114,800**

[1] Intel Core i7-3520M 2.89GHz (Ivy Bridge).
[2] Intel Core i7-2600 3.4GHz (Sandy Bridge).
[3] Intel Core i7-2720QM 2.2GHz (Sandy Bridge).

conclude that the τ-and-add method is more efficient than the halve-and-add, but the former technique suffers from the relatively limited extension fields available for Koblitz curves, which at least for the 128-bit security level case, forces to have larger field elements and thus more expensive field multiplications.

The GLS elliptic curve over a prime field reported in [36] requires 33% more field multiplications than our code. Nevertheless, it benefits from a highly efficient native multiplication with carry instruction (MUL), which allows to generate a fast scalar multiplication. The same observation can be extended to protected implementations when comparing between prime and binary curves.

5.2.3 Faster native multiplication

The Haswell family of processors was launched in 2013, including among other features, the AVX2 set of vector instructions and a faster carry-less multiplier latency and throughput. The latency of this multiplier, compared to previous micro-architectures, was reduced from between 12 and 14 cycles to only 7 cycles, while the reciprocal throughput was reduced from between 7 and 8 cycles to only 2 cycles [14]. In Table 8 we report our timings in this platform, specifically in an Intel Core i7 4770K 3.50GHz machine with HyperThreading and TurboBoost disabled.

When compared with the Sandy Bridge results (see Table 6), the Haswell timings are about 39% faster for the halve-and-add method and about 48% and 50% faster for the protected and unprotected double-and-add implementations, respectively. Note that the faster carry-less multiplication plays the main role in the new results. As a consequence, methods that use more field multiplications, which is the case of the double-and-add, benefit the most. The competitiveness between the double-and-add and halve-and-add methods favors the parallel version, which can almost achieve a two-factor speed-up. When executed in the Haswell platform, the two-core 2-GLV method is 43% faster than the Sandy Bridge timings.

5.2.4 Memory requirements

The library presented in this work is intended for its application in high-end platforms where, typically, memory is an abundant resource. Accordingly, several arithmetic operations aggressively use precomputed tables with the aim of achieving a faster computation than what could be obtained by a direct calculation (See §2 for a detailed discussion).

In particular, the base field implementation of the half-trace operation, uses a precomputed table of size $2^8 \cdot \lceil \frac{m}{8} \rceil$ field elements. Using $m = 128$, this translates to a 2^{16}-byte table. The faster field inverse implementation invokes four multi-squaring operations, but the constant-time implementations uses slower consecutive squarings. Each one of these multi-squaring operations

Table 7 A comparison of several elliptic curve libraries by their required number of field multiplications.

Implementation	Field	Method	Estimated Mult.		Field Mult. cost (cc)
			pre/post	sc. mult.	
Taverne et al. [46]	$\mathbb{F}_{2^{233}}$	No-GLV	92	638	100
Aranha et al. [4]	$\mathbb{F}_{2^{283}}$	2-GLV	100	**572**	142
Longa and Sica [36]	$\mathbb{F}_{p^2}$	4-GLV	113	1004	**80**
This work	$\mathbb{F}_{2^{254}}$	2-GLV	**86**	752	94

Table 8 Timings and memory requirements for scalar multiplication in the Haswell platform, assuming that the input and output points are provided in λ-affine coordinates.

Scalar multiplication	Curve	Security	Method	TAR	Cycles	Memory (bytes)
This work	GLS	127	2-GLV (double-and-add) (λ)	no	**46,700**	$2^{15}+4\times 64$
			2-GLV (halve-and-add) (λ)	no	**42,100**	$2^{16}+2^{15}+4\times 96$
			2-GLV, parallel (2 cores) (λ)	no	**27,300**	$2^{16}+2^{15}+4\times(96+64)$
			2-GLV (double-and-add) (λ)	yes	**60,000**	8×64

requires to precompute a table of size $2^4\cdot\lceil\frac{m}{4}\rceil$ field elements, that translates to a table with a size of 2^{13} bytes. Therefore, the memory cost associated to the faster field inversion computation in our library is of 2^{15} Bytes. Finally, the halve-and-add scalar multiplication requires the storage of 4 accumulators in projective coordinates; and the double-and-add scalar multiplication requires the storage of 4 and 8 multiples of the base point for the unprotected and protected versions, respectively. A summary of the memory costs associated to the scalar multiplication algorithms presented in this work are reported in the last column of Table 8.

6 Conclusion

In this work, a new projective coordinate system that leads to fast elliptic curve operations, was presented. The use of the λ-projective coordinates in combination with an optimized implementation of quadratic field arithmetic and the endomorphisms available in the GLS curves, allowed us to achieve record timings in the scalar multiplication computation for different point configurations, including the fastest protected/unprotected reported computation of kP at 127 bits of security.

In the near future, we expect to use the Haswell AVX2 256-bit extension set along with the bit manipulation instructions for optimizing the quadratic field arithmetic and the scalar multiplication functions even further.

Acknowledgements We wish to thank Sanjit Chatterjee, Patrick Longa and Alfred Menezes for their useful discussions.

References

1. Agnew, G.B., Mullin, R.C., Vanstone, S.A.: An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. IEEE J. Sel. Areas Commun. 11(5), 804–813 (1993)
2. Ahmadi, O., Hankerson, D., Rodríguez-Henríquez, F.: Parallel formulations of scalar multiplication on Koblitz curves. J. UCS 14(3), 481–504 (2008)
3. Al-Daoud, E., Mahmod, R., Rushdan, M., Kilicman, A.: A new addition formula for elliptic curves over $GF(2^n)$. IEEE Trans. Comput. 51(8), 972–975 (2002)
4. Aranha, D.F., Faz-Hernández, A., López, J., Rodríguez-Henríquez, F.: Faster Implementation of Scalar Multiplication on Koblitz Curves. In: Hevia, A., Neven, G. (eds.) LATINCRYPT 2012, LNCS, vol. 7533, pp. 177–193. Springer (2012)
5. Aranha, D.F., López, J., Hankerson, D.: Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets. In: Abdalla, M., Barreto, P.S.L.M. (eds.) LATINCRYPT 2010, LNCS, vol. 6212, pp. 144–161. Springer (2010)
6. Avanzi, R. M., Ciet, M., Sica, F.: Faster Scalar Multiplication on Koblitz Curves Combining Point Halving with the Frobenius Endomorphism. In: Bao, F., Deng, R. H., Zhou, J. (ends.) PKC 2004, LNCS, vol. 2947, pp. 28–40. Springer (2004)
7. Bernstein, D.J.: Curve25519: New Diffie-Hellman Speed Records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006, LNCS, vol. 3958, pp. 207–228. Springer (2006)
8. Bernstein, D.J., Lange, T. (eds.): eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to`. Accessed June 6, 2013.
9. Bernstein, D.J., Lange, T., Farashahi, R.: Binary Edwards Curves. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008, LNCS, vol. 5154, pp. 244–265. Springer (2008)
10. Bos, J, W., Kleinjung T., Niederhagen R., Schwabe P.: ECC2K-130 on Cell CPUs. In: Bernstein D. J., Lange T. (eds.), AFRICACRYPT 2010, LNCS, vol. 6055, pp. 225242. Springer (2010)
11. Bos, J, W., Costello, C., Hisil, H., Lauter, K.: Fast Cryptography in Genus 2. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013, LNCS, vol. 7881, pp. 194–210. Springer (2013)
12. Chatterjee, S., Karabina, K., Menezes, A.: A new protocol for the nearby friend problem. In: Parker, M.G. (ed.)

IMACC 2009, LNCS, vol. 5921, pp. 236–251. Springer (2009)
13. Chudnovsky, D.V., Chudnovsky, G.V.: Sequences of numbers generated by addition in formal groups and new primality and factorization tests. Adv. Appl. Math. 7(4), 385 – 434 (1986)
14. Fog, A.: Instruction Tables: List of Instruction Latencies, Throughputs and Micro-operation Breakdowns for Intel, AMD and VIA CPUs. http://www.agner.org/optimize/instruction_tables.pdf, accessed 28 Oct 2013.
15. Faz-Hernández, A., Longa, P., Sanchez, A.H.: Efficient and Secure Algorithms for GLV-Based Scalar Multiplication and their Implementation on GLV-GLS Curves, In: Benaloh, J. (ed.) CT-RSA 2014, To appear.
16. Firasta, M., Buxton, M., Jinbo, P., Nasri, K., Kuo, S.: Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency. White paper, Intel Corporation, http://software.intel.com (2008)
17. Fong, K., Hankerson, D., López, J., Menezes, A.: Field inversion and point halving revisited. IEEE Trans. Comput. 53(8), 1047–1059 (2004)
18. Galbraith, S., Lin, X., Scott, M.: Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves. J. Cryptol. 24, 446–469 (2011)
19. Gallant, R.P., Lambert, R.J., Vanstone, S.A.: Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. In: Kilian, J. (ed.) CRYPTO 2001, LNCS, vol. 2139, pp. 190–200. Springer (2001)
20. Gaudry, P., Hess, F., Smart, N.P.: Constructive and destructive facets of Weil descent on elliptic curves. J. Cryptol. 15, 19–46 (2002)
21. Hamburg, M.: Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Report 2012/309, http://eprint.iacr.org/ (2012)
22. Hankerson, D., Karabina, K., Menezes, A.: Analyzing the Galbraith-Lin-Scott Point Multiplication Method for Elliptic Curves over Binary Fields. IEEE Trans. Comput. 58(10), 1411–1420 (2009)
23. Hankerson, D., Menezes, A., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2003)
24. Hankerson, D., Hernandez, J., Menezes, A.: Software Implementation of Elliptic Curve Cryptography over Binary Fields. In: Koç, Ç.K., Paar, C. (eds.) CHES 2000, LNCS, vol. 1965, pp. 1–24. Springer (2000)
25. Hess, F.: Generalising the GHS Attack on the Elliptic Curve Discrete Logarithm Problem. LMS J. Comput. Math. 7, 167–192 (2004)
26. Intel Corporation: Intel Architecture Instruction Set Extensions Programming Reference, Reference Number: 319433-014. http://software.intel.com (2012)
27. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. Inf. Comput. 78(3), 171–177 (1988)
28. Joye, M., Tunstall, M.: Exponent recoding and regular exponentiation algorithms. In: Preneel, B. (ed.) AFRICACRYPT 2009, LNCS, vol. 5580, pp. 334–349. Springer (2009)
29. Kim, D., Lim, S.: Integer Decomposition for Fast Scalar Multiplication on Elliptic Curves. In: Nyberg, K., Heys, H. (eds.) SAC 2003, LNCS, vol. 2595, pp. 13–20. Springer (2003)
30. Kim, K.H., I., K.S.: A New Method for Speeding Up Arithmetic on Elliptic Curves over Binary Fields. Cryptology ePrint Archive, Report 2007/181, http://eprint.iacr.org/ (2007)
31. King, B.: An Improved Implementation of Elliptic Curves over $GF(2^n)$ when Using Projective Point Arithmetic. In: Vaudenay, S., Youssef, A. (eds.) SAC 2001, LNCS, vol. 2259, pp. 134–150. Springer (2001)
32. Knudsen, E.: Elliptic Scalar Multiplication Using Point Halving. In: Lam, K.Y., Okamoto, E., Xing, C. (eds.) ASIACRYPT99, LNCS, vol. 1716, pp. 135–149. Springer (1999)
33. Knuth, D.E.: The Art of Computer Programming: Seminumerical Algorithms, vol. 2. Addison-Wesley, Boston (1997)
34. Lange, T.: A note on López-Dahab coordinates. Cryptology ePrint Archive, Report 2004/323, http://eprint.iacr.org/ (2006)
35. Lim, C.H., Hwang, H.S.: Speeding up elliptic scalar multiplication with precomputation. In: Song, J. (ed.) ICISC 1999, LNCS, vol. 1787, pp. 102–119. Springer (2000)
36. Longa, P., Sica, F.: Four-Dimensional Gallant-Lambert-Vanstone Scalar Multiplication. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012, LNCS, vol. 7658, pp. 718–739. Springer (2012)
37. Longa, P., Sica, F.: Four-Dimensional Gallant-Lambert-Vanstone Scalar Multiplication. J. Cryptol., To appear (2013)
38. López, J., Dahab, R.: Improved Algorithms for Elliptic Curve Arithmetic in GF(2^n). In: Tavares, S.E., Meijer, H. (eds.) SAC 1998, LNCS, vol. 1556, pp. 201–212. Springer (1998)
39. López, J., Dahab, R.: An overview of elliptic curve cryptography. Tech. Rep. IC-00-10, Institute of computing, University of Campinas, http://www.ic.unicamp.br/~reltech/2000/00-10.pdf (2000)
40. López, J., Dahab, R.: New Point Compression Algorithms for Binary Curves. In: IEEE Information Theory Workshop (ITW 2006), pp. 126–130, IEEE Press, New York (2006)
41. Park, Y.H., Jeong, S., Kim, C., Lim, J.: An Alternate Decomposition of an Integer for Faster Point Multiplication on Certain Elliptic Curves. In: Naccache, D., Paillier, P. (eds.) PKC 2002, LNCS, vol. 2274, pp. 323–334. Springer (2002)
42. Rodríguez-Henríquez, F., Morales-Luna G., López J.: Low-Complexity Bit-Parallel Square Root Computation over GF(2^m) for All Trinomials. IEEE Trans. Computers 57(4): 472-480 (2008)
43. Schroeppel, R.: Cryptographic elliptic curve apparatus and method (2000), U.S. patent 2002/6490352 B1
44. Schroeppel, R.: Elliptic curve point halving wins big (2000), In: 2nd Midwest Arithmetical Geometry in Cryptography Workshop.
45. Schroeppel, R.: Automatically solving equations in finite fields (2002), U.S. patent 2002/0055962 A1
46. Taverne, J., Faz-Hernández, A., Aranha, D.F., Rodríguez-Henríquez, F., Hankerson, D., López, J.: Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction. Journal of Cryptographic Engineering 1, 187–199 (2011)

A Proofs

Proof of Theorem 1. *Let $P = (x_P, \lambda_P)$ be an elliptic point in $E_{a,b}(\mathbb{F}_{2^m})$. Then a formula for $2P = (x_{2P}, \lambda_{2P})$ is given by*

$$x_{2P} = \lambda_P^2 + \lambda_P + a$$

$$\lambda_{2P} = \frac{x_P^2}{x_{2P}} + \lambda_P^2 + a + 1.$$

From [23], page 81, we have the formulas: $x_{2P} = \lambda_P^2 + \lambda_P + a$ *and* $y_{2P} = x_P^2 + \lambda_P x_{2P} + x_{2P}$. *Then, a formula for* λ_{2P} *can be obtained as follows:*

$$\begin{aligned}
\lambda_{2P} &= \frac{y_{2P} + x_{2P}^2}{x_{2P}} = \frac{(x_P^2 + \lambda_P \cdot x_{2P} + x_{2P}) + x_{2P}^2}{x_{2P}} \\
&= \frac{x_P^2}{x_{2P}} + \lambda_P + 1 + x_{2P} = \frac{x_P^2}{x_{2P}} + \lambda_P + 1 + (\lambda_P^2 + \lambda_P + a) \\
&= \frac{x_P^2}{x_{2P}} + \lambda_P^2 + a + 1.
\end{aligned}$$

In affine coordinates, the doubling formula requires one division and two squarings. Given the point $P = (X_P, L_P, Z_P)$ *in the* λ*-projective representation, an efficient projective doubling algorithm can be derived by applying the doubling formula to the affine point* $(\frac{X_P}{Z_P}, \frac{L_P}{Z_P})$. *For* x_{2P} *we have:*

$$\begin{aligned}
x_{2P} &= \frac{L_P^2}{Z_P^2} + \frac{L_P}{Z_P} + a = \frac{L_P^2 + L_P \cdot Z_P + a \cdot Z_P^2}{Z_P^2} \\
&= \frac{T}{Z_P^2} = \frac{T^2}{T \cdot Z_P^2}.
\end{aligned}$$

For λ_{2P} *we have:*

$$\begin{aligned}
\lambda_{2P} &= \frac{\frac{X_P^2}{Z_P^2}}{\frac{T}{Z_P^2}} + \frac{L_P^2}{Z_P^2} + a + 1 \\
&= \frac{X_P^2 \cdot Z_P^2 + T \cdot (L_P^2 + (a+1) \cdot Z_P^2)}{T \cdot Z_P^2}.
\end{aligned}$$

From the λ*-projective equation, we have the relation* $T \cdot X_P^2 = X_P^4 + b \cdot Z_P^4$. *Then the numerator* w *of* λ_{2P} *can also be written as follows,*

$$\begin{aligned}
w &= X_P^2 \cdot Z_P^2 + T \cdot (L_P^2 + (a+1) \cdot Z_P^2) \\
&= X_P^2 \cdot Z_P^2 + T \cdot L_P^2 + T^2 + T^2 + (a+1) \cdot Z_{2P} \\
&= X_P^2 \cdot Z_P^2 + T \cdot L_P^2 + L_P^4 + L_P^2 \cdot Z_P^2 + a^2 \cdot Z_P^4 + T^2 \\
&+ (a+1) \cdot Z_{2P} \\
&= X_P^2 \cdot Z_P^2 + T \cdot (L_P^2 + X_P^2) + X_P^4 + b \cdot Z_P^4 + L_P^4 \\
&+ L_P^2 \cdot Z_P^2 + a^2 \cdot Z_P^4 + T^2 + (a+1) \cdot Z_{2P} \\
&= (L_P^2 + X_P^2) \cdot ((L_P^2 + X_P^2) + T + Z_P^2) + T^2 \\
&+ (a^2 + b) \cdot Z_P^4 + (a+1) \cdot Z_{2P}.
\end{aligned}$$

This completes the proof.

Proof of Theorem 2. *Let* $P = (x_P, \lambda_P)$ *and* $Q = (x_Q, \lambda_Q)$ *be elliptic points in* $E_{a,b}(\mathbb{F}_{2^m})$. *Then a formula for* $P + Q = (x_{P+Q}, \lambda_{P+Q})$ *is given by*

$$\begin{aligned}
x_{P+Q} &= \frac{x_P \cdot x_Q}{(x_P + x_Q)^2}(\lambda_P + \lambda_Q) \\
\lambda_{P+Q} &= \frac{x_Q \cdot (x_{P+Q} + x_P)^2}{x_{P+Q} \cdot x_P} + \lambda_P + 1.
\end{aligned}$$

Since P *and* Q *are elliptic points on a non-supersingular curve, we have the following relation:* $y_P^2 + x_P \cdot y_P + x_P^3 + a \cdot x_P^2 = b = y_Q^2 + x_Q \cdot y_Q + x_Q^3 + a \cdot x_Q^2$. *The known formula for computing the* x*-coordinate of* $P + Q$ *is given by* $x_{P+Q} = s^2 + s + x_P + x_Q + a$, *where* $s = \frac{y_P + y_Q}{x_P + x_Q}$. *Then one can derive the new formula as follows,*

$$\begin{aligned}
x_{P+Q} &= \frac{(y_P + y_Q)^2 + (y_P + y_Q) \cdot (x_P + y_Q)}{(x_P + x_Q)^2} \\
&+ \frac{(x_P + x_Q)^3 + a \cdot (x_P + x_Q)^2}{(x_P + x_Q)^2} \\
&= \frac{b + b + x_Q \cdot (x_P^2 + y_P) + x_P \cdot (x_Q^2 + y_Q)}{(x_P + x_Q)^2} \\
&= \frac{x_P \cdot x_Q \cdot (\lambda_P + \lambda_Q)}{(x_P + x_Q)^2}.
\end{aligned}$$

For computing λ_{P+Q}, *we use the observation that the* x*-coordinate of* $(P + Q) - P$ *is* x_Q. *We also know that for* $-P$ *we have* $\lambda_{-P} = \lambda_P + 1$ *and* $x_{-P} = x_P$. *By applying the formula for the* x*-coordinate of* $(P + Q) + (-P)$ *we have*

$$\begin{aligned}
x_Q = x_{(P+Q)+(-P)} &= \frac{x_{P+Q} \cdot x_{-P}}{(x_{P+Q} + x_{-P})^2} \cdot (\lambda_{P+Q} + \lambda_{-P}) \\
&= \frac{x_{P+Q} \cdot x_P}{(x_{P+Q} + x_P)^2} \cdot (\lambda_{P+Q} + \lambda_P + 1).
\end{aligned}$$

Then $\lambda_{P+Q} = \frac{x_Q \cdot (x_{P+Q} + x_P)^2}{x_{P+Q} \cdot x_P} + \lambda_P + 1$.

To obtain a λ*-projective addition formula, we apply the formulas above to the affine points* $(\frac{X_P}{Z_P}, \frac{L_P}{Z_P})$ *and* $(\frac{X_Q}{Z_Q}, \frac{L_Q}{Z_Q})$. *Then, the* x_{P+Q} *coordinate of* $P + Q$ *can be computed as:*

$$\begin{aligned}
x_{P+Q} &= \frac{\frac{X_P}{Z_P} \cdot \frac{X_Q}{Z_Q} \cdot (\frac{L_P}{Z_P} + \frac{L_Q}{Z_Q})}{(\frac{X_P}{Z_P} + \frac{X_Q}{Z_Q})^2} \\
&= \frac{X_P \cdot X_Q \cdot (L_P \cdot Z_Q + L_Q \cdot Z_P)}{(X_P \cdot Z_Q + X_Q \cdot Z_P)^2} = X_P \cdot X_Q \cdot \frac{A}{B}.
\end{aligned}$$

For the λ_{P+Q} *coordinate of* $P + Q$ *we have:*

$$\begin{aligned}
\lambda_{P+Q} &= \frac{\frac{X_Q}{Z_Q} \cdot (\frac{X_P \cdot X_Q \cdot A}{B} + \frac{X_P}{Z_P})^2}{\frac{X_P \cdot X_Q \cdot A}{B} \cdot \frac{X_P}{Z_P}} + \frac{L_P + Z_P}{Z_P} \\
&= \frac{(A \cdot X_Q \cdot Z_P + B)^2 + (A \cdot B \cdot Z_Q)(L_P + Z_P)}{A \cdot B \cdot Z_P \cdot Z_Q}.
\end{aligned}$$

In order that both x_{P+Q} *and* λ_{P+Q} *have the same denominator, the formula for* x_{P+Q} *can be written as*

$$X_{P+Q} = \frac{X_P \cdot X_Q \cdot A}{B} = \frac{A \cdot (X_P \cdot Z_Q) \cdot (X_Q \cdot Z_P) \cdot A}{A \cdot B \cdot Z_P \cdot Z_Q}.$$

Therefore, $x_{P+Q} = \frac{X_{P+Q}}{Z_{P+Q}}$ *and* $\lambda_{P+Q} = \frac{L_{P+Q}}{Z_{P+Q}}$. *This completes the proof.*

Proof of Theorem 3. *The* λ*-projective formula is obtained by adding the* λ*-affine points* $2Q = (\frac{X_{2Q}}{Z_{2Q}}, \frac{L_{2Q}}{Z_{2Q}})$ *and* $P = (x_P, \lambda_P)$ *with the formula of Theorem 2. Then, the* x *coordinate of* $2Q + P$ *is given by*

$$\begin{aligned}
x_{2Q+P} &= \frac{x_{2Q} \cdot x_P}{(x_{2Q} + x_P)^2}(\lambda_{2Q} + \lambda_P) \\
&= \frac{X_{2Q} \cdot x_P(L_{2Q} + \lambda_P \cdot Z_{2Q})}{(X_{2Q} + x_P \cdot Z_{2Q})^2} \\
&= \frac{x_P \cdot (X_Q^2 \cdot Z_Q^2 + T \cdot (L_Q^2 + (a + 1 + \lambda_P) \cdot Z_Q^2))}{(T + x_P \cdot Z_Q^2)^2} \\
&= x_P \cdot \frac{A}{B}.
\end{aligned}$$

The λ_{2Q+P} *coordinate of* $2Q+P$ *is computed as*

$$\begin{aligned}\lambda_{2Q+P} &= \frac{\frac{X_{2Q}}{Z_{2Q}} \cdot (x_P \cdot \frac{A}{B} + x_P)^2}{x_P \cdot \frac{A}{B} \cdot x_P} + \lambda_P + 1 \\ &= \frac{T \cdot (A+B)^2 + (\lambda_P + 1) \cdot (A \cdot B \cdot Z_Q^2)}{A \cdot B \cdot Z_Q^2}.\end{aligned}$$

The formula for x_{2Q+P} *can be written with denominator* Z_{2Q+P} *as follows,*

$$x_{2Q+P} = \frac{x_P \cdot A}{B} = \frac{x_P \cdot Z_Q^2 \cdot A^2}{A \cdot B \cdot Z_Q^2}.$$

Therefore, $x_{2Q+P} = \frac{X_{2Q+P}}{Z_{2Q+P}}$ *and* $\lambda_{2Q+P} = \frac{L_{2Q+P}}{Z_{2Q+P}}$*. This completes the proof.*

B Operation count for 2-GLV double-and-add using λ-coordinates

Basically, three cases can occur in the 2-GLV double-and-add main loop. The first one, when the digits of both scalars k_1, k_2 equal zero, we just perform a point doubling (D) in the accumulator. The second one, when both scalar digits are different from zero, we have to double the accumulator and sum two points. In this case, we perform one doubling and addition (DA) followed by a mixed addition (A). Finally, it is possible that just one scalar has its digit different from zero. Here, we double the accumulator and add a point, which can be done with only one doubling-and-addition operation.

Then, as the nonzero bit distributions in the scalars represented by the w-NAF are independent, we have for the first case,

$$Pr[k_{1,i} = 0 \wedge k_{2,i} = 0] = \frac{w^2}{(w+1)^2}, \text{ for } i \in [0, n-1].$$

For the second case,

$$Pr[k_{1,i} \neq 0 \wedge k_{2,i} \neq 0] = \frac{1}{(w+1)^2}, \text{ for } i \in [0, n-1].$$

And for the third case,

$$Pr[(k_{1,i} \neq 0 \wedge k_{2,i} = 0) \vee (k_{1,i} = 0 \wedge k_{2,i} \neq 0)] = \frac{2w}{(w+1)^2}.$$

Consequently, the operation count can be written as

$$\begin{aligned}&\frac{n}{2}\left(\frac{w^2}{(w+1)^2}D + \frac{1}{(w+1)^2}(DA + A) + \frac{2w}{(w+1)^2}DA\right) \\ &= \frac{(2w+1)n}{2(w+1)^2}DA + \frac{w^2 n}{2(w+1)^2}D + \frac{n}{2(w+1)^2}A.\end{aligned}$$

C Parameters used for the Galbraith-Lin-Scott elliptic curve

Using the notation given in §4, let $q = 2^m$, with $m = 127$. The towering of the fields $\mathbb{F}_q$ and its quadratic extension $\mathbb{F}_{q^2} \cong \mathbb{F}_q[u]/(g(u))$ are constructed by means of the irreducible trinomials $f(x) = x^{127} + x^{63} + 1$ and $g(u) = u^2 + u + 1$, respectively. Let $E/\mathbb{F}_q : y^2 + xy = x^3 + ax^2 + b$, with $a, b \in \mathbb{F}_q$, be a binary elliptic curve. and define the quadratic twist of E as the Galbraith-Lin-Scott elliptic curve

$$\tilde{E}/\mathbb{F}_{q^2} : y^2 + xy = x^3 + a'x^2 + b,$$

with $a' \in \mathbb{F}_{q^2}$ such that $Tr(a') = 1$. Given $\#E(\mathbb{F}_q) = q + 1 - t$, it follows that $\#\tilde{E}_{a',b}(\mathbb{F}_{q^2}) = (q-1)^2 + t^2 = h \cdot r$, where t is the trace of Frobenius of the curve E, $h = 2$ and r is 253-bit prime number.

In this work, the binary GLS elliptic curve $\tilde{E}_{a',b}(\mathbb{F}_{q^2})$ was defined with the following parameters

- $a' = u$
- $b \in \mathbb{F}_q$ is a degree 126 binary polynomial that can be represented in hexadecimal format as,
  $b = 0x59C8202CB9E6E0AE2E6D944FA54DE7E5$
- The 253-bit prime order r of the main subgroup of $\tilde{E}_{a',b}(\mathbb{F}_{q^2})$ is,

$$\begin{aligned}r =&0x1FFFFFFFFFFFFFFFFFFFFFFFFFFFFF \\ &FFDAC40D1195270779877DABA2A44750A5;\end{aligned}$$

- The base point $P = (x_p, \lambda_p)$ of order r specified in $\lambda-$affine coordinates is,

$$\begin{aligned}x_p &= 0x203B6A93395E0432344038B63FBA32DE \\ &\quad + 0x78E51FD0C310696D5396E0681AA10E0D \cdot u \\ \lambda_p &= 0x5BD7653482085F55DEB59C6137074B50 \\ &\quad + 0x7F90D98B1589A17F24568FA5A1033946 \cdot u.\end{aligned}$$