Faster implementation of scalar multiplication on Koblitz curves

Diego F. Aranha[1], Armando Faz-Hernández[2],
Julio López[3], and Francisco Rodríguez-Henríquez[2]

[1] Departament of Computer Science, University of Brasília
dfaranha@unb.br
[2] Computer Science Department, CINVESTAV-IPN
armfaz@computacion.cs.cinvestav.mx, francisco@cs.cinvestav.mx
[3] Institute of Computing, University of Campinas
jlopez@ic.unicamp.br

Abstract. We design a state-of-the-art software implementation of field and elliptic curve arithmetic in standard Koblitz curves at the 128-bit security level. Field arithmetic is carefully crafted by using the best formulae and implementation strategies available, and the increasingly common native support to binary field arithmetic in modern desktop computing platforms. The i-th power of the Frobenius automorphism on Koblitz curves is exploited to obtain new and faster interleaved versions of the well-known τNAF scalar multiplication algorithm. The usage of the $\tau^{\lfloor m/3 \rfloor}$ and $\tau^{\lfloor m/4 \rfloor}$ maps are employed to create analogues of the 3-and 4-dimensional GLV decompositions and in general, the $\lfloor m/s \rfloor$-th power of the Frobenius automorphism is applied as an analogue of an s-dimensional GLV decomposition. The effectiveness of these techniques is illustrated by timing the scalar multiplication operation for fixed, random and multiple points. To our knowledge, our library was the first to compute a random point scalar multiplication in less than 10^5 clock cycles among all curves with or without endomorphisms defined over binary or prime fields. The results of our optimized implementation suggest a trade-off between speed, compliance with the published standards and side-channel protection. Finally, we estimate the performance of curve-based cryptographic protocols instantiated using the proposed techniques and compare our results to related work.

Key words: Efficient software implementation, Koblitz elliptic curves, scalar multiplication.

1 Introduction

Since its introduction in 1985, Elliptic Curve Cryptography (ECC) has become one of the most important and efficient public key cryptosystems in use. Its security is based on the computational intractability of solving discrete logarithm problems over the group formed by the rational points on an elliptic curve.

Anomalous binary curves, also known as Koblitz elliptic curves, were introduced in [1]. Since then, these curves have been subject of extensive analysis and study. Given a finite field $\mathbb{F}_q$ for $q = 2^m$, a Koblitz curve $E_a(\mathbb{F}_q)$, is defined as the set of points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ that satisfy the equation

$$E_a : y^2 + xy = x^3 + ax^2 + 1, \quad a \in \{0, 1\}, \tag{1}$$

together with a point at infinity denoted by $\mathcal{O}$. It is known that $E_a(\mathbb{F}_q)$ forms an additive Abelian group with respect to the elliptic point addition operation. In this paper, E_a is a Koblitz curve with order $\#E_a(\mathbb{F}_{2^m}) = 2^{2-a}r$, where r is an odd prime. Let $\langle P \rangle$ be an additively written subgroup in E_a of prime order r, and let k be a positive integer such that $k \in [0, r-1]$. Then, the elliptic curve scalar multiplication operation computes the multiple $Q = kP$, which corresponds to the point resulting of adding P to itself, $k-1$ times. Given r, P and $Q \in \langle P \rangle$, the Elliptic Curve Discrete Logarithm Problem (ECDLP) consists of finding the unique integer k such that $Q = kP$ holds.

Since Koblitz curves are defined over the binary field $\mathbb{F}_2$, the Frobenius map and its inverse naturally extend to an automorphism of the curve denoted by τ. The τ map takes (x, y) to (x^2, y^2) and $\mathcal{O}$ to $\mathcal{O}$. It can been shown that $(x^4, y^4) + 2(x, y) = \mu(x^2, y^2)$ for every (x, y) on E_a, where $\mu = (-1)^{1-a}$. In other words, τ satisfies $\tau^2 + 2 = \mu\tau$. By solving the quadratic equation, we can associate τ with the complex number $\tau = \frac{-1+\sqrt{-7}}{2}$.

Elliptic curve scalar multiplication is the most expensive operation in cryptographic protocols whose security guarantees are based on the ECDLP. Improving the computational efficiency of this operation is a widely studied problem. Across the years, a number of algorithms and techniques providing efficient implementations with higher performance have been proposed [2]. Many research works have focused their efforts on the unknown point scenario, where the base point P is not known in advance and when only one single scalar multiplication is required, as in the case of the Diffie-Hellman key exchange protocol [3,4,5]. However, there are situations where a single scalar multiplication must be performed on fixed base points such as in the case of the key and signature generation procedures of the Elliptic Curve Digital Signature Algorithm (ECDSA) standard. In other scenarios, such as in the ECDSA signature verification, the simultaneous computation of two scalar multiplications (one with unknown point and the other with fixed point) of the form $R = kG + lQ$, is required. Comparatively less research works have studied the latter cases [6,7,8].

In [9,3], authors evaluated the achievable performance of binary elliptic curve arithmetic in the latest 64-bit micro-architectures, presenting a comprehensive analysis of unknown-point scalar multiplication computations on random and Koblitz NIST elliptic curves at the 112-bit and 192-bit security levels. However, for the 128-bit security level they only considered a random curve with side-channel resistant scalar multiplication.[4] This was mainly due to the unavail-

[4] Scalar multiplication on curve CURVE2251 was implemented in [3] using the Montgomery laddering approach that is naturally protected against first-order side-channel attacks.

ability of benchmarking data for curves equipped with endomorphisms and the performance penalty of halving-based approaches when applied to standardized curves.

In this work we revisit the software serial computation of scalar multiplication on Koblitz curves defined over binary fields. This study includes the computation of the scalar multiplication using unknown and fixed points; and single and simultaneous scalar multiplication computations as required in the generation and verification of discrete-log based digital signatures. We extend the analysis given in [3,9] and further investigate an alternate curve choice to provide a complete picture of the performance scenario, while also showing through operation counting and experimental results that Koblitz curves are still the fastest choice for deploying curve-based cryptography if sufficient native support for binary field arithmetic is available in the target platform and if resistance to software side-channel attacks can be disregarded.

To this end, we adopted several techniques previously proposed by different authors: (i) formulation of binary field arithmetic using vector instructions [10]; (ii) time-memory trade-offs for the evaluation of fixed 2^k-powers in binary fields [11]; (iii) new formulas for polynomial multiplication over $\mathbb{F}_2$ and its extensions [12]; (iv) efficient support for the recently introduced carry-less multiplier [3].

Besides building on these advancements on finite field arithmetic, this paper presents several novel techniques including: (i) improved implementation of width-w τNAF integer recoding; (ii) a new precomputation scheme for small multiples of a random point in a Koblitz curve; (iii) lazy-reduction formulae for mixed addition in binary elliptic curves; (iv) novel interleaving strategies of the τNAF algorithm for scalar multiplication in Koblitz curves via powers of the Frobenius automorphism. We remark that the interleaved techniques proposed in this work can be seen as the effective application for the first time in Koblitz curves of an s-dimensional GLV decomposition. Moreover, in this work only the "tried and tested" Koblitz curve NIST-K283 is considered, providing immediate compatibility and interoperability with standards and existing implementations. Note, however, that several of our techniques are not restricted in any sense to this curve choice, and can therefore be used to accelerate scalar multiplication in other Koblitz curves at different security levels.

Our main implementation result is a speed record for the unknown-point single-core scalar multiplication computation over the NIST-K283 curve in a little less than 10^5 clock cycles. Running on an Intel Core i7-2600K processor clocked at 3.4 GHz, we were able to compute a random point scalar multiplication in just 29.18μs.

This document is structured as follows: Section 2 discusses the low-level techniques used for the implementation of field arithmetic and integer recoding. Section 3 presents high-level techniques for arithmetic in the elliptic curve, comprising improved formulas for mixed addition by means of lazy reduction and strategies for speeding up the scalar multiplication computation by using powers of the Frobenius automorphism. Section 4 illustrates the efficiency of the pro-

posed techniques reporting operation counts and timings for scalar multiplication in the fixed, unknown and multiple point scenarios; and extensively compares the results with related work. Additionally in this section we estimate the performance of signature and key agreement protocols when they are instantiated with Koblitz curves. The final section concludes the paper with perspectives for further performance improvement based on upcoming instruction sets.

2 Low-level techniques

Let $f(z)$ be a monic irreducible polynomial of degree m over $\mathbb{F}_2$. Then, the binary extension field $\mathbb{F}_{2^m}$ is isomorphic to $\mathbb{F}_{2^m} \cong \mathbb{F}_2[z]/(f(z))$, i.e., $\mathbb{F}_{2^m}$ is a finite field of characteristic 2, whose elements are the finite set of all the binary polynomials of degree less than m. In order to achieve a security level equivalent to 128-bit AES when working with binary elliptic curves, NIST recommends to choose the field extension $\mathbb{F}_{2^{283}}$, along with the irreducible pentanomial $f(z) = z^{283} + z^{12} + z^7 + z^5 + 1$. In a modern 64-bit computing platform, an element from the field $\mathbb{F}_{2^m}$ represented in canonical basis requires $n_{64} = \lceil \frac{m}{64} \rceil$ processor words, or $n_{64} = 5$ when $m = 283$. In the rest of this section, descriptions of algorithms and formulas will refer to either generic or fixed versions of the binary field, depending on whether or not the optimization is restricted to the choice of $m = 283$.

As mentioned before, in this work we made an extensive use of vector instruction sets present in contemporary desktop processors. The platform model given in Table 1 extends the notation reported in [10]. There is limited support for flexible bitwise shifting in vector registers, because propagation of bits between the two contiguous 64-bit words requires additional operations. Notice that vectorized multiple-precision or intra-digit shifts can always be made faster when the shift amount is a multiple of 8 by means of the memory alignment instruction or the bytewise shift instruction, respectively, and that a simultaneous table lookup mapping 4-bit indexes to bytes can be implemented through the byte shuffling instruction called `PSHUFB` in the SSE instruction set.

Table 1: Relevant vector instructions for the implementation of binary field arithmetic.

Mnemonic	Description	SSE
$\otimes$	Carry-less multiplication	`PCLMULQDQ`
$\ll_{\dagger 8}, \gg_{\dagger 8}$	64-bit bitwise shifts	`PSLLQ`,`PSRLQ`
$\ll_8, \gg_8$	128-bit bytewise shift	`PSLLDQ`,`PSRLDQ`
$\oplus, \wedge, \vee$	Bitwise `XOR`,`AND`,`OR`	`PXOR`,`PAND`,`POR`
$\lhd, \rhd$	Memory alignment/Multi-precision shifts	`PALIGNR`

In the following, we provide brief implementation notes on how relevant field arithmetic operations such as, addition, multiplication, squaring, multi-squaring, modular reduction and inversion; and integer width-w τNAF recoding, were implemented.

Addition. It is the simplest operation in a binary field and can employ the exclusive-or instruction with the largest operand size in the target platform.

This is particularly beneficial for vector instructions, but according to our experiments, the 128-bit SSE [13] integer instruction proved to be faster than the 256-bit AVX [14] floating-point instruction due to a higher reciprocal throughput [15] when operands are stored into registers.

Multiplication. Field multiplication is the performance-critical arithmetic operation for elliptic curve arithmetic. Given two field elements $a(z), b(z) \in \mathbb{F}_{2^{283}}$ we want to compute a third field element $c(z) = a(z) \cdot b(z) \bmod f(z)$. This can be accomplished by performing two separate steps: first the polynomial multiplication of the two operands $a(z), b(z)$ is evaluated and then the resulting double length polynomial is modular reduced by $f(z)$. From our field element representation, the polynomial multiplication step can be seen as the computation of the product of two $(n_{64} - 1)$-degree polynomials, each with n_{64} 64-bit coefficients. Alternatively, the two operands may also be seen as $(\lceil \frac{n_{64}}{2} \rceil - 1)$-degree polynomials, each with $\lceil \frac{n_{64}}{2} \rceil$ 128-bit coefficients. In the latter case, each term-by-term multiplication can be solved via the standard Karatsuba formula by performing 3 carry-less multiplications. When $n_{64} = 5$, the above approaches require 13 (see [12,16]) and 14 invocations of the carry-less multiplier instruction, respectively. Algorithm 1 below presents our implementation of field multiplication over the field $\mathbb{F}_{2^{283}}$ with 64-bit granularity using the formula given in [12]. The computational complexity of Algorithm 1 is of 13 carry-less multiplications and 32 vector additions, respectively, plus one modular reduction (Alg. 1, step 22) that will be discussed later. The most salient feature of Algorithm 1 is that all the 13 carry-less multiplications have been grouped into one single loop on steps 6-8. This is an attractive feature from a throughput point of view, as it is important to potentially reduce the cost of the carry-less multiplication instruction from 14 to 8 clock cycles in the Intel Sandy Bridge micro-architecture; and from 12 to 7 clock cycles in an AMD Bulldozer [15]. The rationale behind this cost reduction is that the batch execution of independent multiplications directly benefits the micro-architecture pipeline occupancy level. It is worth mentioning that in [3], authors concluded that the 64-bit granular approach tends to consume more resources and complicate register allocation, limiting the natural throughput exhibited by the carry-less multiplication instruction. However, if the digits are stored in an interleaved form (see [17]), these side effects are mitigated and higher throughput can again be achieved.

Squaring and multi-squaring. Squaring is a cheap operation in a binary field due to the action of the Frobenius map, consisting of a linear expansion of coefficients. Vectorized approaches using simultaneous table lookups through byte shuffling instructions allow a particularly efficient formulation of the coefficient expansion step [10]. Modular reduction usually is the most expensive step when computing a squaring, especially when $f(z)$ is an *ordinary pentanomial* (see [18]) for the word size. Dealing efficiently with ordinary pentanomials requires flexible and often not directly supported shifting instructions in the target platform. Multi-squaring is a time-memory trade-off

in which a table of $16\lceil \frac{m}{4} \rceil$ field elements allows computing any fixed 2^k power with the cost equivalent of just a few squarings [11]. It is usually the case that the multi-squaring approach becomes faster than repeated squaring, whenever $k \geq 6$ [3]. Contrary to addition, the availability of 256-bit instructions here contributes significantly to a performance increase. This happens because this operation basically consists of a sequence of additions with field elements obtained through a precomputed table stored in main memory.

Algorithm 1 Proposed implementation of multiplication in $\mathbb{F}_{2^{283}}$.

Input: $a(z) = a[0..4], b(z) = b[0..4]$.
Output: $c(z) = c[0..4] = a(z) \cdot b(z)$.
Note: Pairs a_i, b_i, c_i, m_i of 64-bit words represent vector registers.

1: **for** $i \leftarrow 0$ **to** 4 **do**
2: $\quad c_i \leftarrow (a[i], b[i])$
3: **end for**
4: $c_5 \leftarrow c_0 \oplus c_1, \quad c_6 \leftarrow c_0 \oplus c_2, \quad c_7 \leftarrow c_2 \oplus c_4, \quad c_8 \leftarrow c_3 \oplus c_4$
5: $c_9 \leftarrow c_3 \oplus c_6, \quad c_{10} \leftarrow c_1 \oplus c_7, \quad c_{11} \leftarrow c_5 \oplus c_8, \quad c_{12} \leftarrow c_2 \oplus c_{11}$
6: **for** $i \leftarrow 0$ **to** 12 **do**
7: $\quad m_i \leftarrow c_i[0] \otimes c_i[1]$
8: **end for**
9: $c_0 \leftarrow m_0, \quad c_8 \leftarrow m_4$
10: $c_1 \leftarrow c_0 \oplus m_1, \quad c_2 \leftarrow c_1 \oplus m_6$
11: $c_1 \leftarrow c_1 \oplus m_5, \quad c_2 \leftarrow c_2 \oplus m_2$
12: $c_7 \leftarrow c_8 \oplus m_3, \; c_6 \leftarrow c_7 \oplus m_7$
13: $c_7 \leftarrow c_7 \oplus m_8, \; c_6 \leftarrow c_6 \oplus m_2$
14: $c_5 \leftarrow m_{11} \oplus m_{12}, c_3 \leftarrow c_5 \oplus m_9$
15: $c_3 \leftarrow c_3 \oplus c_0 \oplus c_{10}$
16: $c_4 \leftarrow c_1 \oplus c_7 \oplus m_9 \oplus m_{10} \oplus m_{12}$
17: $c_5 \leftarrow c_5 \oplus c_2 \oplus c_8 \oplus m_{10}$
18: $c_9 \leftarrow c_7 \ll_8 64$
19: $(c_7, c_5, c_3, c_1) \leftarrow (c_7, c_5, c_3, c_1) \lhd 8$
20: $c_0 \leftarrow c_0 \oplus c_1, \quad c_1 \leftarrow c_2 \oplus c_3, \quad c_2 \leftarrow c_4 \oplus c_5$
21: $c_3 \leftarrow c_6 \oplus c_7, \quad c_4 \leftarrow c_8 \oplus c_9$
22: **return** $c = (c_4, c_3, c_2, c_1, c_0) \bmod f(z)$

Modular reduction. Efficient modular reduction of a double-length value resulting of a squaring or multiplication operation to a proper field element involves expressing the required shifted additions in terms of the best shifting instructions possible. For the instruction sets available in our target platform, this amounts to converting the highest possible number of shifts to memory alignment instructions or byte-wise shifts. Curve NIST-K283 is defined over an ordinary pentanomial, a particularly inefficient choice for our vector register size. However, by observing that $f(z) = z^{283} + z^{12} + z^7 + z^5 + 1 = z^{283} + (z^7 + 1)(z^5 + 1)$, one can take advantage of this factorization to formulate faster shifted additions. Algorithm 2 presents our explicit scheduling

of shift instructions to perform modular reduction in $\mathbb{F}_{2^{283}}$. Suppose that the polynomial c is written as $c = p_1||p_0$ where the polynomial p_0 represent the lower 283 bits of c. The computation of $c \bmod f(z)$ in Algorithm 2 is performed as follows: in lines 1 to 3, the polynomial p_1 is computed by shifting the vector (c_4, c_3, c_2) to the right exactly 27 bits. Then, in lines 4 to 10, the operation $c + p_1(z^7 + 1)(z^5 + 1)$ is performed, thus getting the vector (c_2, c_1, c_0) . Finally, in lines 11 to 14, the remaining 101 most significant bits of c_2 are reduced, a process that again involves a multiplication by the polynomial $(z^7 + 1)(z^5 + 1)$.

Algorithm 2 Implementation of reduction by $f(z) = z^{283} + (z^7 + 1)(z^5 + 1)$.

Input: Double-precision polynomial stored into 128-bit registers $c = (c_4, c_3, c_2, c_1, c_0)$.
Output: Field element $c \bmod f(z)$ stored into 128-bit registers (c_2, c_1, c_0).

1: $t_2 \leftarrow c_2, t_0 \leftarrow (c_3, c_2) \rhd 64, t_1 \leftarrow (c_4, c_3) \rhd 64$
2: $c_4 \leftarrow c_4 \gg_{\sharp 8} 27, c_3 \leftarrow c_3 \gg_{\sharp 8} 27, c_3 \leftarrow c_3 \oplus (t_1 \ll_{\sharp 8} 37)$
3: $c_2 \leftarrow c_2 \gg_{\sharp 8} 27, c_2 \leftarrow c_2 \oplus (t_0 \ll_{\sharp 8} 37)$
4: $t_0 \leftarrow (c_4, c_3) \rhd 120, c_4 \leftarrow c_4 \oplus (t_0 \gg_{\sharp 8} 1)$
5: $t_1 \leftarrow (c_3, c_2) \rhd 64, c_3 \leftarrow c_3 \oplus (c_3 \ll_{\sharp 8} 7) \oplus (t_1 \gg_{\sharp 8} 57)$
6: $t_0 \leftarrow c_2 \ll_8 64, c_2 \leftarrow c_2 \oplus (c_2 \ll_{\sharp 8} 7) \oplus (t_0 \gg_{\sharp 8} 57)$
7: $t_0 \leftarrow (c_4, c_3) \rhd 120, c_4 \leftarrow c_4 \oplus (t_0 \gg_{\sharp 8} 3)$
8: $t_1 \leftarrow (c_3, c_2) \rhd 64, c_3 \leftarrow c_3 \oplus (c_3 \ll_{\sharp 8} 5) \oplus (t_1 \gg_{\sharp 8} 59)$
9: $t_0 \leftarrow c_2 \ll_8 64, c_2 \leftarrow c_2 \oplus (c_2 \ll_{\sharp 8} 5) \oplus (t_0 \gg_{\sharp 8} 59)$
10: $c_0 \leftarrow c_0 \oplus c_2, c_1 \leftarrow c_1 \oplus c_3, c_2 \leftarrow t_2 \oplus c_4$
11: $t_0 \leftarrow c_4 \gg_{\sharp 8} 27$
12: $t_1 \leftarrow t_0 \oplus (t_0 \ll_{\sharp 8} 5)$
13: $t_0 \leftarrow t_1 \oplus (t_1 \ll_{\sharp 8} 7)$
14: $c_0 \leftarrow c_0 \oplus t_0, c_2 \leftarrow c_2 \wedge$ (`0x0000000000000000,0x0000000007FFFFFF`)
15: **return** $c = (c_2, c_1, c_0)$

Inversion. The field inversion approach that probably is the friendliest to vector instruction sets is the Itoh-Tsuji inversion [19] that computes the field inverse of a using the identity $a^{-1} = \left(a^{2^{m-1}-1}\right)^2$. The term $a^{2^{m-1}-1}$ is obtained by sequentially computing intermediate terms of the form

$$\left(a^{2^i-1}\right)^{2^j} \cdot a^{2^j-1}. \tag{2}$$

where the exponents $0 \leq i, j \leq m-1$, are elements of the addition chain associated to the exponent $e = m-1$ [20,21]. The shortest addition chain for $e = 282$ has length 11 and is $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 17 \rightarrow 34 \rightarrow 35 \rightarrow 70 \rightarrow 140 \rightarrow 141 \rightarrow 282$. The computation of the above outlined procedure introduces an important memory cost of storing 4 multi-squaring tables (for computing powers $2^{17}, 2^{35}, 2^{70}, 2^{141}$), with each table containing $16\lceil \frac{m}{4} \rceil$ field elements. However, several of those tables can be reused in the interleaving approach for scalar

multiplication by exploiting powers of the Frobenius automorphism as will be explained in the next section. We note that other approaches for computing multiplicative field inverses, such as a polynomial version of the extended euclidean algorithm, tend to be not so efficient when vectorized mostly because they require intensive shifting of the intermediate values generated by the algorithm.

Integer τNAF recoding Solinas [22] presented a τ-adic analogue of the customary *Non-Adjacent Form* (NAF) recoding. An element $\rho \in \mathbb{Z}[\tau]$ is found with $\rho \equiv k \pmod{\frac{\tau^m-1}{\tau-1}}$, of as small norm as possible, where for the subgroup of interest, $kP = \rho P$ and a width-w τNAF representation for ρ can be obtained in a way that mimics the usual width-w NAF recoding. As in [22], let us define $\alpha_i = i \bmod \tau^w$ for $i \in \{1, 3, 5, \ldots, 2^{w-1} - 1\}$. A width-$w$ τNAF of a nonzero element k is an expression $k = \sum_{i=0}^{l-1} u_i \tau^i$ where each $u_i \in \{0, \pm\alpha_1, \pm\alpha_3, \ldots, \pm\alpha_{2^{w-1}-1}\}$ and $u_{l-1} \neq 0$, and at most one of any consecutive w coefficients is nonzero. Under reasonable assumptions, this procedure outputs an expansion with length $l \leq m+1$. Although the cost of width-w NAF recoding is usually negligible when compared with the overall cost of scalar multiplication, this is not generally the case with Koblitz curves, where integer to width-w τNAF recoding can reach more than 10% of the computational time for computing a scalar multiplication [3]. In this work, the recoding was implemented by employing as much as possible branchless techniques: the branches inside the recoding operation essentially depend on random values, presenting a worst-case scenario for branch prediction and causing severe performance penalties. In addition to that, the code was also completely unrolled to handle only the precision required in the current iteration. Since the magnitude of the involved scalars gets reduced with each iteration, it is suboptimal to perform operations considering the initial full precision. The deterministic nature of the algorithm allows one to know in which precise iteration of the main recoding loop, the most significant word of the intermediate values become zero, which permits to represent these values with one less processor word.

3 High-level techniques

In the last section, several notes gave a general description of our algorithmic and implementation choices for field arithmetic. This section describes the higher-level strategies used in the elliptic curve arithmetic layer for increasing the performance of scalar multiplication.

3.1 Exploiting powers of the Frobenius automorphism

Scalar multiplication algorithms on Koblitz curves are always tailored to exploit the Frobenius automorphism τ on $E(\mathbb{F}_{2^m})$ given by $\tau(x, y) = (x^2, y^2)$. One such example is the classic τNAF scalar multiplication algorithm [22] and its width-w

window variants. Given $k \in \mathbb{Z}$ and $P \in E(\mathbb{F}_{2^m})$, these methods work by first writing $k = \sum k_i \tau^i$ for $k_i \in \{0, \pm\alpha_1, \pm\alpha_3, \ldots, \pm\alpha_{2^{w-1}-1}\}$, with $\alpha_i = i \bmod \tau^w$ for $i \in \{1, 3, 5, \ldots, 2^{w-1} - 1\}$. Then the scalar multiplication is computed as $kP = \sum k_i \tau^i P$.

While powers τ^i of the automophism can be automatically considered endomorphisms in the context of the GLV method [23], this does not bring any performance improvement, since applying these powers to a point has exactly the same cost of iterating the automorphism during a standard execution of the τNAF algorithm. Nevertheless, by employing time-memory trade-offs for computing fixed 2^i-th powers with cost significantly smaller than i consecutive squarings, a map of the form $\tau^{\lfloor m/i \rfloor}$ can now be seen as an endomorphism useful for accelerating scalar multiplication through interleaving strategies. For example, the map $\psi \equiv \tau^{\lfloor m/2 \rfloor}$ allows an interleaved scalar multiplication of two points from the expression $kP = k_1 P + 2^{\lfloor m/2 \rfloor} k_2 P = \sum k_{1,i} \tau^i P + \sum k_{2,i} \tau^i \psi(P)$, saving the computational cost of $\lfloor \frac{m}{2} \rfloor$ applications of the Frobenius, or $3\lfloor \frac{m}{2} \rfloor$ squarings. This might be seen as a modest saving, since squaring in a binary field is often considered a free of cost operation. However, this is not entirely true when working with cumbersome irreducible polynomials that lead to relatively expensive modular reductions. This is exactly the case studied in this work and, to be more precise, it can be said instead that interleaving via the ψ endomorphism saves the computational cost associated to $3\lfloor \frac{m}{2} \rfloor$ modular reductions.

As explained above, the map ψ achieves an analogue of a bidimensional GLV decomposition for a Koblitz curve. Similarly, the usage of the $\tau^{\lfloor m/3 \rfloor}$ and $\tau^{\lfloor m/4 \rfloor}$ maps can be seen as analogues to 3- and 4-dimensional GLV decompositions or, more generally, the $\lfloor m/s \rfloor$-th power of the Frobenius automorphism as an analogue of an s-dimensional GLV decomposition. In our working case where $m = 283$, note that the addition chain for Itoh-Tsuji inversion was already chosen to include $\lfloor m/2 \rfloor$ and $\lfloor m/4 \rfloor$. Thus, exploiting these powers of the automorphism does not imply additional storage costs. Observe that [24,9] already explored this concept to obtain parallel formulations of scalar multiplication in Koblitz curves.

3.2 Lazy-reduced mixed point addition

The fastest formula for the mixed addition $R = (X_3, Y_3, Z_3)$ of points $P = (X_1, Y_1, Z_1)$ and $Q = (X_2, Y_2)$ in binary curves use López-Dahab coordinates [25] and were proposed in [26]. When the a-coefficient of the curve is 0, the formula is given below:

$$\begin{gathered} A = Y_1 + Y_2 \cdot Z_1^2, \quad B = X_1 + X_2 \cdot Z_1, \quad C = B \cdot Z_1 \\ Z_3 = C^2, \quad D = X_2 \cdot Z_3, \quad E = A \cdot C \\ X_3 = E + (A^2 + C \cdot B^2), \quad Y_3 = (D + X_3) \cdot (E + Z_3) + (Y_2 + X_2) \cdot Z_3^2. \end{gathered}$$

Evaluating this formula has a cost of 8 field multiplications, 5 field squarings and 8 additions. It is possible to further save 2 modular reductions when computing sums of products in the expressions for the coordinates X_3 and Y_3

given above. This technique is called lazy reduction [27] and trades off a modular reduction by a double-length addition. Our working case presents the best conditions for lazy reduction due to the poor choice of the irreducible pentanomial associated to the NIST K-283 elliptic curve, and the high computational efficiency of the field addition operation. It is then possible to evaluate the formula with a cost equivalent to 8 unreduced multiplications, 5 unreduced squarings, 11 modular reductions, and 10 field addditions. This is very similar to the formula proposed in [28], but without introducing any new coordinates to chain unreduced values across sequential additions.

3.3 Scalar multiplication algorithm

Algorithm 3 provides a generic interleaved version of the width-w τNAF point multiplication method when the main loop is folded s times by exploring the $\lfloor m/s \rfloor$-th power of the Frobenius automorphism. In comparison with the original algorithm, approximately $3(s-1)\lfloor \frac{m}{s} \rfloor$ field squarings are saved. Notice however, that incrementing the value s also increases the computational and storage costs of constructing the table of base-point multiples performed in Steps 2-5. In the following, the construction of this table of points is referred as *precomputation phase*.

3.4 Precomputation scheme

The scalar multiplication algorithm presented in Algorithm 3 requires the computation of the set of affine points $P_{0,u} = \alpha_u P$, for $u \in \{1, 3, 5, \ldots, 2^{w-1} - 1\}$. Basically, there are two simple approaches to compute this set: use inversion-free addition in projective coordinates and convert all the points at the end to affine coordinates using the Montgomery's simultaneous inversion method; or perform the additions directly in affine coordinates. High inversion-to-multiplication ratios clearly favor the former approach. The latter can be made more viable when the ratio is moderate and simultaneous inversion is employed for computing the denominators in affine addition.

For an illustration of both approaches, assume the choice $w = 5$, and let M, S, A, I be the cost of multiplication, squaring, addition and inversion in $\mathbb{F}_{2^m}$, respectively. Let us consider first the strategy of performing most of the operations in projective coordinates. For the selected value of w, the first four point multiples of the precomputation table given as,

$$\alpha_1 P = P;\ \alpha_3 P = (\tau^2 - 1)P;\ \alpha_5 P = (\tau^2 + 1)P;\ \alpha_7 P = (\tau^3 - 1)P;$$

can be computed in projective coordinates at a cost of three point additions plus three Frobenius operations. However, the last 4 point multiples in the table, namely,

$$\alpha_9 P = (\tau^3 \alpha_5 + 1)P;\ \ \alpha_{11} P = (-\tau^2 \alpha_5 - 1)P;$$
$$\alpha_{13} P = (-\tau^2 \alpha_5 + 1)P;\ \alpha_{15} P = (-\tau^2 \alpha_5 - \alpha_5)P;$$

Algorithm 3 Interleaved width-w τNAF scalar multiplication using $\tau^{\lfloor m/s \rfloor}$.

Input: $k \in \mathbb{Z}, P \in E(\mathbb{F}_{2^m})$, integer s denoting the interleaving factor.
Output: $kP \in E(\mathbb{F}_{2^m})$.

Compute width-w τ-NAF$(k) = \sum_{i=0}^{l-1} u_i \tau^i$
Compute $P_{0,u} = \alpha_u P$, for $u \in \{1, 3, 5, \ldots, 2^{w-1} - 1\}$
for $i \leftarrow 1$ **to** $(s-1)$ **do**
    Compute $P_{i,u} = \tau^{\lfloor m/s \rfloor} P_{i-1,u}$
end for
$Q \leftarrow \infty$
for $i \leftarrow l-1$ **to** $s\lfloor \frac{m}{s} \rfloor$ **do**
    $Q \leftarrow \tau Q$
    if $u_i \neq 0$ **then**
        Let u be such that $\alpha_u = u_i$ or $\alpha_{-u} = -u_i$
        if $u_i > 0$ **then** $Q \leftarrow Q + P_{0,u}$; **else** $Q \leftarrow Q - P_{0,u}$
    end if
end for
for $i \leftarrow (\lfloor \frac{m}{s} \rfloor - 1)$ **to** 0 **do**
    $Q \leftarrow \tau Q$
    for $j \leftarrow 0$ **to** $(s-1)$ **do**
        if $u_{i+j\lfloor m/s \rfloor} \neq 0$ **then**
            Let u be such that $\alpha_u = u_{i+j\lfloor m/s \rfloor}$ or $\alpha_{-u} = -u_{i+j\lfloor m/s \rfloor}$
            if $u_i > 0$ **then** $Q \leftarrow Q + P_{j,u}$; **else** $Q \leftarrow Q - P_{j,u}$
        end if
    end for
end for
return $Q = (x, y)$

can be only computed until the point $\tau^2 \alpha_5 P$ has been calculated [2]. This situation requires either an expensive conversion to affine coordinates of the point $\tau^2 \alpha_5 P$ or the lower penalty of performing one general instead of a mixed point addition with an associated cost of $(13M + 4S + 9A)$. Hence, it is possible to compute all the required points with just 6 point additions or subtractions, a single general point addition, 6 Frobenius in affine or projective coordinates and a simultaneous conversion of 7 points to affine coordinates. Half of the 6 point additions and subtractions mentioned above are between points in affine coordinates and considering the associated cost of simultaneous Montgomery inversion, each of them has a computational cost of just $(5M + 3S + 8A)$ and one single inversion. Hence, the total precomputation cost for $w = 5$ is given as,

$$\begin{aligned}\text{Proj. Precomputation cost} &= 3 \cdot (5M + 3S + 8A) + 3 \cdot (8M + 5S + 8A) + \\ &\quad 3 \cdot 2S + 3 \cdot 3S + (13M + 4S + 9A) + \\ &\quad 3 \cdot (7-1)M + I + 7 \cdot (2M + S) \\ &= 84M + 50S + 57A + I.\end{aligned}$$

On the other hand, let us consider the second approach where all the additions are directly performed in affine coordinates. Let us recall that one affine addition

costs $2M + S + I + 8A$. Due to the dependency previously mentioned, we have to split all the affine addition computations into two groups $\{\alpha_3 P, \alpha_5 P, \alpha_7 P\}$ and $\{\alpha_9 P, \alpha_{11} P, \alpha_{13} P, \alpha_{15} P\}$, without dependencies. Computing the first group requires 3 affine additions and a simultaneous inversion to obtain 3 line slopes; whereas the second group requires 4 affine additions and a simultaneous inversion to obtain the 4 slopes, for a total of $7 \cdot (2M + S + 8A) + 3(3-1)M + 3(4-1)M + 2I = 29M + 7S + 56A + 2I$. Considering only the dominant multiplications and inversions, the affine precomputation scheme will be faster than the projective precomputation scheme whenever the inversion-to-multiplication ratio is lower than 55, an assumption entirely compatible with the target platform [3].

4 Estimates, results and discussion

4.1 Performance estimates

Now we are in a position to estimate the performance of Algorithm 3 for the values of $m = 283, s = 1, w = 5$. The algorithm executes the precomputation scheme described in the last section, an average of m applications of the Frobenius automorphism, an expected number of $\frac{m}{w+1}$ additions and a final conversion to affine coordinates. This amounts to a cost of about,

$$\begin{aligned} \text{Estimated cost of Algorithm 3} &= 29M + 7S + 56A + 2I + 283 \cdot 3S + \\ &\quad 47 \cdot (8M + 5S + 8A) + (I + 2M + S) \\ &= 407M + 1092S + 3I \end{aligned}$$

For comparison, the current state-of-the-art serial implementation of a random point multiplication, using a 4-dimensional GLV method over a prime curve and the same choice of w, takes 1 inversion, 742 multiplications, 225 squarings and 767 additions in $\mathbb{F}_{p^2}$, where p has approximately 128 bits [29]. By using the latest formula for 5-term polynomial multiplication described in the last section, the scalar multiplication in Koblitz curves is expected to execute $407 \cdot 13 = 5291$ word multiplications, while the GLV-capable prime curve is expected to execute $(742 \cdot 3 + 225 \cdot 2) \cdot 4 = 10704$ word multiplications. This rough comparison means that a scalar multiplication in a Koblitz curve should be considerably faster than a prime curve equipped with endomorphisms if sufficient support to binary field multiplication is present, or even twice faster if this support is equivalent to integer multiplication. Although the latency of the fastest carry-less multiplier available (7 cycles at best [15]) is substantially higher than the integer multiplier counterpart (3 cycles [15]), from our analysis above, it is still entirely possible that a careful implementation of a Koblitz curve comparable computational cost.

4.2 Experimental results

In order to illustrate the performance obtained by the proposed techniques, we implemented a library targeted to the Intel Westmere and Sandy Bridge

micro-architectures, focusing our efforts on benefitting from the SSE and AVX instruction sets with the corresponding availability of 128-bit and 256-bit registers. The library was implemented in the C programming language, with vector instructions accessed through their intrinsics interface. Both version 4.7.1 of the GNU C Compiler Suite (GCC) and version 12.1 of the Intel C Compiler (ICC) were used to build the library in a GNU/Linux environment.

Benchmarking was conducted on Intel Core i5-540M and Core i7-2600K processors clocked at 2.5GHz and 3.4 GHz, respectively, following the guidelines provided in the EBACS website [30]. Namely, automatic overclocking, frequency scaling and HyperThreading technologies were disabled to reduce randomness in the results.

Table 2 presents timings and ratios related to the cost of multiplication for the low-level field arithmetic layer of the library, which computes basic operations in the field $\mathbb{F}_{2^{283}}$. Note how modular reduction dominates the cost of squaring and how the moderate inversion-to-multiplication ratios justify the algorithmic choices. Our best timing on Sandy Bridge for unreduced multiplication is 5% faster than the 135 cycles reported in [31], this saving is obtained by a careful implementation of the same polynomial multiplication formula used in [31].

Table 2: Timings given in clock cycles for basic operations in $\mathbb{F}_{2^{283}}$.

	Westmere			Sandy Bridge		
Base field operation	GCC	ICC	op/M	GCC	ICC	op/M
Modular reduction	28	28	0.11	20	22	0.15
Unreduced multiplication	159	163	0.89	128	132	0.89
Multiplication	182	182	1.00	142	149	1.00
Squaring	42	39	0.21	28	29	0.18
Multi-Squaring	287	295	1.62	235	243	1.63
Inversion	4,372	4,268	23.45	3,286	3,308	22.20

Table 3 shows the number of clock cycles for elliptic curve operations, such as point addition, Frobenius endomorphism, and point doubling. The latter is shown only to reflect the improvement of using point doubling-free scalar multiplication as is the case in Koblitz curves. Integer recoding is almost 3 times faster than [3,9], even with longer scalars.

Timings reported for scalar multiplication are divided into three scenarios: (i) known point, where the point to be multiplied is already known before the execution of scalar multiplication; (ii) unknown point, the general case, where the input point is not known until scalar multiplication is processed; (iii) double multiplication of a fixed and a random point, a case usually needed for verifying curve-based digital signatures. For the three scenarios, we used interleaved versions of the left-to-right width-w window τNAF scalar multiplication algorithm with different choices of w. We present timings in Table 4. It was verified experimentally that $s = 2$ is the best choice for random and double point multi-

Table 3: Elliptic curve operations on NIST-K283 when points are represented in affine or López-Dahab coordinates [25].

	Westmere			Sandy Bridge		
Elliptic curve operation	GCC	ICC	op/M	GCC	ICC	op/M
Frobenius (Affine)	84	70	0.38	55	55	0.37
Frobenius (LD)	118	115	0.63	85	83	0.55
Doubling (LD)	965	939	5.15	741	764	5.12
Addition (LD Mixed)	1,684	1,650	9.06	1,300	1,336	8.96
Addition (LD General)	2,683	2,643	14.52	2,086	2,145	14.39
Width-w τNAF recoding	4,841	6,652	36.55	3,954	4,693	31.50

plication, providing a speedup of 3-5% over the conventional case $s = 1$, and that $s = 4$ provides a significant performance increase for fixed point multiplication.

Table 4: Scalar multiplication in three different scenarios: fixed, random and multiple points. Timings are given in 10^3 processing cycles.

	Westmere		Sandy Bridge	
Scalar multiplication	GCC	ICC	GCC	ICC
Random point (kP), $w = 5, s = 1$	139.6	135.1	105.3	105.3
Random point (kP), $w = 5, s = 2$	130.9	127.8	99.2	99.7
Fixed point (kG), $w = 8, s = 2$	80.8	79.0	61.5	62.3
Fixed point (kG), $w = 8, s = 4$	72.6	71.7	55.1	55.9
Fixed/random point ($kG + lQ$), $w_G = 6, w_Q = 5, s = 2$	207.8	206.8	157.7	160.8
Fixed/random point ($kG + lQ$), $w_G = 8, w_Q = 5, s = 2$	192.3	190.6	146.3	148.7

4.3 Comparison to related work

The current state-of-the-art is an implementation by Longa and Sica at the 128-bit security level on a Sandy Bridge platform and achieves an unprotected scalar multiplication of a random point on a prime curve in 91,000 clock cycles with 16 precomputed points; and a side-channel resistant scalar multiplication in 137,000 cycles with 36 precomputed points [29]. A protected implementation by Bernstein et al. [8] reports 226,872 cycles for computing this operation on Westmere and 194,208 cycles on Sandy Bridge [30]. Another implementation by Hamburg [32] reports 153,000 cycles on Sandy Bridge. Our implementation is only 9% slower than the current speed record when computing instances of the ECDH key agreement protocol, even with considerably lower platform support for the underlying field arithmetic.

Computing curve-based digital signatures usually amounts to scalar multiplication of fixed points. The authors of [8] report a latency of 87,548 cycles to compute this operation on the Westmere and 70,292 cycles on the Sandy

Bridge [30] micro-architectures, while using a precomputed table of 256 points. Hamburg [32] implemented this operation on Sandy Bridge in just 52,000 cycles with 160 precomputed points. Compared to the first implementation and using the same number of points, our timings are faster by 22%. Comparing to the second implementation while reducing the number of precomputed points to 128, our timings are slower by 15%.

The last scenario to analyze is signature verification, where work [8] reports single signature verification timings of 273,364 cycles on Westmere and 226,516 cycles on Sandy Bridge [30], while reporting significantly improved timings for batch verification. A faster implementation [32] verifies a signature using 32 precomputed points on Sandy Bridge in 165,000 cycles. We obtain speedups between 5% and 35% on this scenario, considering implementations with the same number of points, and leave the possibility of batch verification as a future direction of this work. It is important to stress that our implementation provides a trade-off between side-channel protection and standards compliance. Consequently, it allows faster and interoperable curve-based cryptography when resistance to side channels is not required.

5 Conclusion

In this work, we presented a software implementation of elliptic curve arithmetic in Koblitz curves defined over binary fields. By reusing several low-level techniques recently-introduced by other authors and proposing a number of useful high-level techniques, we obtained state-of-the-art timings for computing scalar multiplication of a random point in a binary curve, modelling a curve-based key agreement protocol. Our implementation also provides a trade-off between execution time and storage overhead for computing digital signatures and significantly improves the time to verify a single signature. We expect our timings to be accelerated further as support to binary field arithmetic improves on modern 64-bit platforms, either through a faster carry-less multiplier or via the 256-bit integer vector instructions from the upcoming AVX2 instruction set. Our computational cost analysis suggests that if the target platform had a binary field multiplication instruction as efficient as integer multiplication, our implementation could still receive a further factor-2 speedup.

References

1. Koblitz, N.: CM-Curves with Good Cryptographic Properties. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 279–287. Springer (1991)
2. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer-Verlag, Secaucus, USA (2003)
3. 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**(3) 187–199 (2011)

4. Longa, P., Gebotys, C.H.: Efficient techniques for high-speed elliptic curve cryptography. In Mangard, S., Standaert, F.X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 80–94. Springer (2010)
5. Gaudry, P., Thomé, E.: The mpFq library and implementing curve-based key exchanges. In: Software Performance Enhancement of Encryption and Decryption (SPEED 2007), pp. 49–64. http://www.hyperelliptic.org/SPEED/record.pdf (2009)
6. Brown, M., Hankerson, D., López, J., Menezes, A.: Software Implementation of the NIST Elliptic Curves Over Prime Fields. In Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 250–265. Springer (2001)
7. Galbraith, S.D., Lin, X., Scott, M.: Endomorphisms for faster elliptic curve cryptography on a large class of curves. In Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 518–535. Springer (2009)
8. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. In Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 124–142. Springer (2011)
9. Taverne, J., Faz-Hernández, A., Aranha, D.F., Rodríguez-Henríquez, F., Hankerson, D., López, J.: Software implementation of binary elliptic curves: Impact of the carry-less multiplier on scalar multiplication. In Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 108–123. Springer (2011)
10. 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.) In LATINCRYPT 2010. LNCS, vol. 6212, pp. 144-161. Springer (2010)
11. Bos, J.W., Kleinjung, T., Niederhagen, R., Schwabe, P.: ECC2K-130 on Cell CPUs. In D, J.B., Lange, T. (eds.) AFRICACRYPT 2010. LNCS, vol. 6055, pp. 225–242. Springer (2010)
12. Cenk, M., Özbudak, F.: Improved Polynomial Multiplication Formulas over $\mathbb{F}_2$ Using Chinese Remainder Theorem. IEEE Trans. Computers **58**(4) 572–576 (2009)
13. Intel: Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference. http://www.intel.com (2002)
14. Firasta, N., Buxton, M., Jinbo, P., Nasri, K., Kuo, S.: Intel AVX: New frontiers in performance improvement and energy efficiency. White paper available at http://software.intel.com/ (2008)
15. 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 (2012)
16. Montgomery, P.: Five, six, and seven-term Karatsuba-like formulae. IEEE Transactions on Computers **54**(3) 362–369 (2005)
17. Gaudry, P., Brent, R., Zimmermann, P., Thomé, E.: The gf2x binary field multiplication library. https://gforge.inria.fr/projects/gf2x/
18. Scott, M.: Optimal Irreducible Polynomials for $GF(2^m)$ Arithmetic. Cryptology ePrint Archive, Report 2007/192. http://eprint.iacr.org/ (2007)
19. 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)
20. Guajardo, J., Paar, C.: Itoh-Tsujii inversion in standard basis and its application in cryptography and codes. Designs, Codes and Cryptography **25**(2) 207–216 (2002)
21. Rodríguez-Henríquez, F., Morales-Luna, G., Saqib, N.A., Cruz-Cortés, N.: Parallel Itoh—Tsujii multiplicative inversion algorithm for a special class of trinomials. Des. Codes Cryptography **45**(1) 19–37 (2007)

22. Solinas, J.A.: Efficient Arithmetic on Koblitz Curves. Designs, Codes and Cryptography **19**(2-3) 195–249 (2000)
23. Gallant, R., Lambert, R., Vanstone, S.: Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. In Kilian, J., (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 190–200. Springer (2001)
24. Ahmadi, O., Hankerson, D., Rodríguez-Henríquez, F.: Parallel formulations of scalar multiplication on Koblitz curves. Journal of Universal Computer Science **14**(3) 481–504 (2008)
25. López, J., Dahab, R.: Improved Algorithms for Elliptic Curve Arithmetic in GF(2^n). In Tavares, S.E., Meijer, H. (eds.) SAC 98. LNCS, vol. 1556, pp. 201–212. Springer (1998)
26. Al-Daoud, E., Mahmod, R., Rushdan, M., Kiliçman, A.: A New Addition Formula for Elliptic Curves over GF(2^{n}). IEEE Trans. Computers **51**(8) 972–975 (2002)
27. Weber, D., Denny, T.F.: The Solution of McCurley's Discrete Log Challenge. In Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 458–471. Springer (1998)
28. Kim, K.H., Kim, S.I.: A new method for speeding up arithmetic on elliptic curves over binary fields. Cryptology ePrint Archive, Report 2007/181. `http://eprint.iacr.org/` (2007)
29. Longa, P., Sica, F.: Four-Dimensional Gallant-Lambert-Vanstone Scalar Multiplication. In ASIACRYPT 2012. To appear. (2012)
30. Bernstein, D.J., (editors), T.L.: eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to`, (May 18, 2012).
31. Su, C., Fan, H.: Impact of Intel's new instruction sets on software implementation of GF(2)[x] multiplication. Inf. Process. Lett. **112**(12) 497–502 (2012)
32. Hamburg, M.: Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Report 2012/309. `http://eprint.iacr.org/` (2012)

A Appendixes

We complete tables 3.9 and 3.10 from [2] to include corresponding values for $w = 7, 8$.

Table 5: Expressions for $\alpha_u = u \mod \tau^w$ for $w = 7$.

u	$u \bmod \tau^w$	TNAF($u \bmod \tau^w$)	α_u
1	1	(1)	1
3	3	(-1, 0, 0, 1, 0, -1)	$-\tau^2\alpha_{39} - 1$
5	5	(-1, 0, 0, 1, 0, 1)	$-\tau^2\alpha_{39} + 1$
7	7	(-1, 0, 1, 0, 0, -1)	$-\tau^3\alpha_{35} - 1$
9	$3\tau - 5$	(1, 0, 0, -1, 0, 1, 0, 0, 1)	$-\tau^3\alpha_3 + 1$
11	$3\tau - 3$	(-1, 0, -1, 0, -1, 0, -1)	$-\tau^2\alpha_{53} - 1$
13	$3\tau - 1$	(-1, 0, -1, 0, -1, 0, 1)	$-\tau^2\alpha_{53} + 1$
15	$3\tau + 1$	(1, 0, 0, 0, -1)	$\tau^2\alpha_{37} - \alpha_{37}$
17	$3\tau + 3$	(1, 0, 0, 0, 1)	$\tau^2\alpha_{35} + \alpha_{37}$
19	$3\tau + 5$	(1, 0, 0, -1, 0, 1, 0, -1)	$-\tau^2\alpha_3 - 1$
21	$-4\tau - 3$	(-1, 0, 1, 0, 1)	$-\tau^2\alpha_{35} + 1$
23	$-4\tau - 1$	(1, 0, 0, -1, 0, 0, -1)	$\tau^3\alpha_{39} - 1$
25	$-4\tau + 1$	(1, 0, 0, -1, 0, 0, 1)	$\tau^3\alpha_{39} + 1$
27	$-4\tau + 3$	(1, 0, 0, 0, -1, 0, -1)	$\tau^2\alpha_{15} - 1$
29	$6\tau + 1$	(-1, 0, 0, 0, -1, 0, 1)	$-\tau^2\alpha_{17} + 1$
31	$-\tau - 7$	(1, 0, 0, 0, 0, -1)	$\tau^2\alpha_{39} + \alpha_{35}$
33	$-\tau - 5$	(1, 0, 0, 0, 0, 1)	$\tau^2\alpha_{39} + \alpha_{37}$
35	$-\tau - 3$	(1, 0, -1)	$\tau^2 - 1$
37	$-\tau - 1$	(1, 0, 1)	$\tau^2 + 1$
39	$-\tau + 1$	(1, 0, 0, -1)	$\tau^3 - 1$
41	$-\tau + 3$	(1, 0, 0, 1)	$\tau^3 + 1$
43	$-\tau + 5$	(1, 0, 1, 0, -1, 0, -1)	$\tau^2\alpha_{51} - 1$
45	$-\tau + 7$	(1, 0, 1, 0, -1, 0, 1)	$\tau^2\alpha_{51} + 1$
47	$2\tau - 5$	(-1, 0, -1, 0, 0, 0, -1)	$-\tau^2\alpha_{53} + \alpha_{35}$
49	$2\tau - 3$	(-1, 0, -1, 0, 0, 0, 1)	$-\tau^2\alpha_{53} + \alpha_{37}$
51	$2\tau - 1$	(1, 0, 1, 0, -1)	$\tau^2\alpha_{37} - 1$
53	$2\tau + 1$	(1, 0, 1, 0, 1)	$\tau^2\alpha_{37} + 1$
55	$2\tau + 3$	(-1, 0, -1, 0, 0, -1)	$-\tau^3\alpha_{37} - 1$
57	$2\tau + 5$	(-1, 0, -1, 0, 0, 1)	$-\tau^3\alpha_{37} + 1$
59	$2\tau + 7$	(-1, 0, 0, -1, 0, -1)	$-\tau^2\alpha_{41} - 1$
61	$-5\tau - 1$	(-1, 0, -1, 0, 0, -1, 0, 1)	$\tau^2\alpha_{55} + 1$
63	$-5\tau + 1$	(1, 0, 0, 0, 0, 0, -1)	$\tau^2\alpha_{15} + \alpha_{35}$

$a = 0$.

u	$u \bmod \tau^w$	TNAF($u \bmod \tau^w$)	α_u
1	1	(1)	1
3	3	(1, 0, 0, 1, 0, -1)	$-\tau^2\alpha_{39} - 1$
5	5	(1, 0, 0, 1, 0, 1)	$-\tau^2\alpha_{39} + 1$
7	7	(1, 0, -1, 0, 0, -1)	$-\tau^3\alpha_{35} - 1$
9	$-3\tau - 5$	(1, 0, 0, 1, 0, -1, 0, 0, 1)	$\tau^3\alpha_3 + 1$
11	$-3\tau - 3$	(-1, 0, -1, 0, -1, 0, -1)	$-\tau^2\alpha_{53} - 1$
13	$-3\tau - 1$	(-1, 0, -1, 0, -1, 0, 1)	$-\tau^2\alpha_{53} + 1$
15	$-3\tau + 1$	(1, 0, 0, 0, -1)	$\tau^2\alpha_{37} - \alpha_{37}$
17	$-3\tau + 3$	(1, 0, 0, 0, 1)	$\tau^2\alpha_{35} + \alpha_{37}$
19	$-3\tau + 5$	(-1, 0, 0, -1, 0, 1, 0, -1)	$-\tau^2\alpha_3 - 1$
21	$4\tau - 3$	(-1, 0, 1, 0, 1)	$-\tau^2\alpha_{35} + 1$
23	$4\tau - 1$	(1, 0, 0, 1, 0, 0, -1)	$-\tau^3\alpha_{39} - 1$
25	$4\tau + 1$	(1, 0, 0, 1, 0, 0, 1)	$-\tau^3\alpha_{39} + 1$
27	$4\tau + 3$	(1, 0, 0, 0, -1, 0, -1)	$\tau^2\alpha_{15} - 1$
29	$-6\tau + 1$	(-1, 0, 0, 0, -1, 0, 1)	$-\tau^2\alpha_{17} + 1$
31	$\tau - 7$	(-1, 0, 0, 0, 0, -1)	$\tau^2\alpha_{39} + \alpha_{35}$
33	$\tau - 5$	(-1, 0, 0, 0, 0, 1)	$\tau^2\alpha_{39} + \alpha_{37}$
35	$\tau - 3$	(1, 0, -1)	$\tau^2 - 1$
37	$\tau - 1$	(1, 0, 1)	$\tau^2 + 1$
39	$\tau + 1$	(-1, 0, 0, -1)	$-\tau^3 - 1$
41	$\tau + 3$	(-1, 0, 0, 1)	$-\tau^3 + 1$
43	$\tau + 5$	(1, 0, 1, 0, -1, 0, -1)	$\tau^2\alpha_{51} - 1$
45	$\tau + 7$	(1, 0, 1, 0, -1, 0, 1)	$\tau^2\alpha_{51} + 1$
47	$-2\tau - 5$	(-1, 0, -1, 0, 0, 0, -1)	$-\tau^2\alpha_{53} + \alpha_{35}$
49	$-2\tau - 3$	(-1, 0, -1, 0, 0, 0, 1)	$-\tau^2\alpha_{53} + \alpha_{37}$
51	$-2\tau - 1$	(1, 0, 1, 0, -1)	$\tau^2\alpha_{37} - 1$
53	$-2\tau + 1$	(1, 0, 1, 0, 1)	$\tau^2\alpha_{37} + 1$
55	$-2\tau + 3$	(1, 0, 1, 0, 0, -1)	$\tau^3\alpha_{37} - 1$
57	$-2\tau + 5$	(1, 0, 1, 0, 0, 1)	$\tau^3\alpha_{37} + 1$
59	$-2\tau + 7$	(1, 0, 0, -1, 0, -1)	$-\tau^2\alpha_{41} - 1$
61	$5\tau - 1$	(1, 0, 1, 0, 0, -1, 0, 1)	$\tau^2\alpha_{55} + 1$
63	$5\tau + 1$	(1, 0, 0, 0, 0, 0, -1)	$\tau^2\alpha_{15} + \alpha_{35}$

$a = 1$.

Table 6: Expressions for $\alpha_u = u \mod \tau^w$ for $w = 8$.

u	$u \bmod \tau^w$	TNAF($u \bmod \tau^w$)	α_u
1	1	(1)	1
3	3	(-1, 0, 0, 1, 0, -1)	$\tau^2\alpha_{89} - 1$
5	5	(-1, 0, 0, 1, 0, 1)	$\tau^2\alpha_{89} + 1$
7	7	(-1, 0, 1, 0, 0, -1)	$\tau^3\alpha_{93} - 1$
9	$3\tau - 5$	(1, 0, 0, -1, 0, 1, 0, 0, 1)	$-\tau^3\alpha_3 + 1$
11	$3\tau - 3$	(-1, 0, -1, 0, -1, 0, -1)	$\tau^2\alpha_{75} - 1$
13	$3\tau - 1$	(-1, 0, -1, 0, -1, 0, 1)	$\tau^2\alpha_{75} + 1$
15	$3\tau + 1$	(1, 0, 0, 0, -1)	$\tau^2\alpha_{93} - \alpha_{93}$
17	$3\tau + 3$	(1, 0, 0, 0, 1)	$\tau^2\alpha_{93} - \alpha_{91}$
19	$3\tau + 5$	(1, 0, 0, -1, 0, 1, 0, -1)	$-\tau^2\alpha_3 - 1$
21	$3\tau + 7$	(1, 0, 0, -1, 0, 1, 0, 1)	$-\tau^2\alpha_3 + 1$
23	$3\tau + 9$	(-1, 0, -1, 0, 0, -1, 0, 0, -1)	$-\tau^3\alpha_{73} - 1$
25	$6\tau - 3$	(-1, 0, 0, -1, 0, 0, 1)	$\tau^3\alpha_{87} + 1$
27	$6\tau - 1$	(-1, 0, 0, 0, -1, 0, -1)	$-\tau^2\alpha_{17} - 1$
29	$6\tau + 1$	(-1, 0, 0, 0, -1, 0, 1)	$-\tau^2\alpha_{17} + 1$
31	$6\tau + 3$	(1, 0, 1, 0, 0, 0, 0, -1)	$-\tau^3\alpha_{75} + \alpha_{87}$
33	$6\tau + 5$	(1, 0, 1, 0, 0, 0, 0, 1)	$-\tau^3\alpha_{75} + \alpha_{89}$
35	$6\tau + 7$	(1, 0, 0, 0, 0, 1, 0, -1)	$-\tau^2\alpha_{95} - 1$
37	$6\tau + 9$	(1, 0, 0, 0, 0, 1, 0, 1)	$-\tau^2\alpha_{95} + 1$
39	$6\tau + 11$	(1, 0, 0, 0, 1, 0, 0, -1)	$\tau^3\alpha_{17} - 1$
41	$-8\tau - 7$	(-1, 0, 0, 0, 1, 0, 0, 1)	$-\tau^3\alpha_{15} + 1$
43	$-8\tau - 5$	(1, 0, 0, -1, 0, 1, 0, -1, 0, -1)	$\tau^2\alpha_{19} - 1$
45	$-8\tau - 3$	(1, 0, 0, -1, 0, 1, 0, -1, 0, 1)	$\tau^2\alpha_{19} + 1$
47	$-8\tau - 1$	(1, 0, -1, 0, 0, 0, -1)	$\tau^2\alpha_{109} + \alpha_{91}$
49	$-8\tau + 1$	(1, 0, -1, 0, 0, 0, 1)	$\tau^2\alpha_{109} + \alpha_{93}$
51	$-5\tau - 11$	(-1, 0, 0, 1, 0, 1, 0, -1)	$\tau^2\alpha_5 - 1$
53	$-5\tau - 9$	(-1, 0, 0, 1, 0, 1, 0, 1)	$\tau^2\alpha_5 + 1$
55	$-5\tau - 7$	(-1, 0, -1, 0, -1, 0, 0, -1)	$\tau^3\alpha_{75} - 1$
57	$-5\tau - 5$	(-1, 0, -1, 0, -1, 0, 0, 1)	$\tau^3\alpha_{75} + 1$
59	$-5\tau - 3$	(-1, 0, -1, 0, 0, -1, 0, -1)	$-\tau^2\alpha_{73} - 1$
61	$-5\tau - 1$	(-1, 0, -1, 0, 0, -1, 0, 1)	$-\tau^2\alpha_{73} + 1$
63	$-5\tau + 1$	(1, 0, 0, 0, 0, 0, -1)	$\tau^2\alpha_{17} + \alpha_{91}$
65	$-5\tau + 3$	(1, 0, 0, 0, 0, 0, 1)	$\tau^2\alpha_{17} + \alpha_{93}$
67	$-2\tau - 9$	(1, 0, 0, 1, 0, -1)	$-\tau^2\alpha_{87} - 1$
69	$-2\tau - 7$	(1, 0, 0, 1, 0, 1)	$-\tau^2\alpha_{87} + 1$
71	$-2\tau - 5$	(1, 0, 1, 0, 0, -1)	$-\tau^3\alpha_{91} - 1$
73	$-2\tau - 3$	(1, 0, 1, 0, 0, 1)	$-\tau^3\alpha_{91} + 1$
75	$-2\tau - 1$	(-1, 0, -1, 0, -1)	$\tau^2\alpha_{91} - 1$
77	$-2\tau + 1$	(-1, 0, -1, 0, 1)	$\tau^2\alpha_{91} + 1$
79	$-2\tau + 3$	(1, 0, 1, 0, 0, 0, -1)	$-\tau^2\alpha_{77} - \alpha_{93}$
81	$-2\tau + 5$	(1, 0, 1, 0, 0, 0, 1)	$-\tau^2\alpha_{77} - \alpha_{91}$
83	$\tau - 7$	(-1, 0, -1, 0, 1, 0, -1)	$\tau^2\alpha_{77} - 1$
85	$\tau - 5$	(-1, 0, -1, 0, 1, 0, 1)	$\tau^2\alpha_{77} + 1$
87	$\tau - 3$	(-1, 0, 0, -1)	$-\tau^3 - 1$
89	$\tau - 1$	(-1, 0, 0, 1)	$-\tau^3 + 1$
91	$\tau + 1$	(-1, 0, -1)	$-\tau^2 - 1$
93	$\tau + 3$	(-1, 0, 1)	$-\tau^2 + 1$
95	$\tau + 5$	(-1, 0, 0, 0, 0, -1)	$\tau^2\alpha_{89} + \alpha_{91}$
97	$\tau + 7$	(-1, 0, 0, 0, 0, 1)	$\tau^2\alpha_{89} + \alpha_{93}$
99	$\tau + 9$	(-1, 0, -1, 0, 0, 0, 1, 0, -1)	$-\tau^2\alpha_{79} - 1$
101	$4\tau - 3$	(-1, 0, 0, 0, 1, 0, 1)	$-\tau^2\alpha_{15} + 1$
103	$4\tau - 1$	(-1, 0, 0, 1, 0, 0, -1)	$\tau^3\alpha_{89} - 1$
105	$4\tau + 1$	(-1, 0, 0, 1, 0, 0, 1)	$\tau^3\alpha_{89} + 1$
107	$4\tau + 3$	(1, 0, -1, 0, -1)	$-\tau^2\alpha_{93} - 1$
109	$4\tau + 5$	(1, 0, -1, 0, 1)	$-\tau^2\alpha_{93} + 1$
111	$4\tau + 7$	(1, 0, 0, -1, 0, 0, 0, -1)	$-\tau^2\alpha_3 + \alpha_{93}$
113	$4\tau + 9$	(1, 0, 0, -1, 0, 0, 0, 1)	$-\tau^2\alpha_3 + \alpha_{91}$
115	$4\tau + 11$	(-1, 0, -1, 0, 1, 0, 1, 0, -1)	$\tau^2\alpha_{85} - 1$
117	$7\tau - 1$	(-1, 0, 1, 0, 1, 0, 1)	$-\tau^2\alpha_{107} + 1$
119	$7\tau + 1$	(1, 0, 1, 0, -1, 0, 0, -1)	$-\tau^3\alpha_{77} - 1$
121	$7\tau + 3$	(1, 0, 1, 0, -1, 0, 0, 1)	$-\tau^3\alpha_{77} + 1$
123	$7\tau + 5$	(1, 0, 1, 0, 0, -1, 0, -1)	$\tau^2\alpha_{71} - 1$
125	$7\tau + 7$	(1, 0, 1, 0, 0, -1, 0, 1)	$\tau^2\alpha_{71} + 1$
127	$7\tau + 9$	(1, 0, 0, 0, 0, 0, 0, -1)	$-\tau^2\alpha_{95} + \alpha_{91}$

$a = 0$.

u	$u \bmod \tau^w$	TNAF($u \bmod \tau^w$)	α_u
1	1	(1)	1
3	3	(1, 0, 0, 1, 0, -1)	$\tau^2\alpha_{89} - 1$
5	5	(1, 0, 0, 1, 0, 1)	$\tau^2\alpha_{89} + 1$
7	7	(1, 0, -1, 0, 0, -1)	$\tau^3\alpha_{93} - 1$
9	$-3\tau - 5$	(1, 0, 0, 1, 0, -1, 0, 0, 1)	$\tau^3\alpha_3 + 1$
11	$-3\tau - 3$	(-1, 0, -1, 0, -1, 0, -1)	$\tau^2\alpha_{75} - 1$
13	$-3\tau - 1$	(-1, 0, -1, 0, -1, 0, 1)	$\tau^2\alpha_{75} + 1$
15	$-3\tau + 1$	(1, 0, 0, 0, -1)	$\tau^2\alpha_{93} - \alpha_{93}$
17	$-3\tau + 3$	(1, 0, 0, 0, 1)	$\tau^2\alpha_{93} - \alpha_{91}$
19	$-3\tau + 5$	(-1, 0, 0, -1, 0, 1, 0, -1)	$-\tau^2\alpha_3 - 1$
21	$-3\tau + 7$	(-1, 0, 0, -1, 0, 1, 0, 1)	$-\tau^2\alpha_3 + 1$
23	$-3\tau + 9$	(-1, 0, -1, 0, 0, 1, 0, 0, -1)	$\tau^3\alpha_{73} - 1$
25	$-6\tau - 3$	(-1, 0, 0, 1, 0, 0, 1)	$-\tau^3\alpha_{87} + 1$
27	$-6\tau - 1$	(-1, 0, 0, 0, -1, 0, -1)	$-\tau^2\alpha_{17} - 1$
29	$-6\tau + 1$	(-1, 0, 0, 0, -1, 0, 1)	$-\tau^2\alpha_{17} + 1$
31	$-6\tau + 3$	(-1, 0, -1, 0, 0, 0, 0, -1)	$\tau^3\alpha_{75} + \alpha_{87}$
33	$-6\tau + 5$	(-1, 0, -1, 0, 0, 0, 0, 1)	$\tau^3\alpha_{75} + \alpha_{89}$
35	$-6\tau + 7$	(-1, 0, 0, 0, 0, 1, 0, -1)	$-\tau^2\alpha_{95} - 1$
37	$-6\tau + 9$	(-1, 0, 0, 0, 0, 1, 0, 1)	$-\tau^2\alpha_{95} + 1$
39	$-6\tau + 11$	(-1, 0, 0, 0, -1, 0, 0, -1)	$-\tau^3\alpha_{17} - 1$
41	$8\tau - 7$	(1, 0, 0, 0, -1, 0, 0, 1)	$\tau^3\alpha_{15} + 1$
43	$8\tau - 5$	(-1, 0, 0, -1, 0, 1, 0, -1, 0, -1)	$\tau^2\alpha_{19} - 1$
45	$8\tau - 3$	(-1, 0, 0, -1, 0, 1, 0, -1, 0, 1)	$\tau^2\alpha_{19} + 1$
47	$8\tau - 1$	(1, 0, -1, 0, 0, 0, -1)	$\tau^2\alpha_{109} + \alpha_{91}$
49	$8\tau + 1$	(1, 0, -1, 0, 0, 0, 1)	$\tau^2\alpha_{109} + \alpha_{93}$
51	$5\tau - 11$	(1, 0, 0, 1, 0, 1, 0, -1)	$\tau^2\alpha_5 - 1$
53	$5\tau - 9$	(1, 0, 0, 1, 0, 1, 0, 1)	$\tau^2\alpha_5 + 1$
55	$5\tau - 7$	(1, 0, 1, 0, 1, 0, 0, -1)	$-\tau^3\alpha_{75} - 1$
57	$5\tau - 5$	(1, 0, 1, 0, 1, 0, 0, 1)	$-\tau^3\alpha_{75} + 1$
59	$5\tau - 3$	(1, 0, 1, 0, 0, -1, 0, -1)	$-\tau^2\alpha_{73} - 1$
61	$5\tau - 1$	(1, 0, 1, 0, 0, -1, 0, 1)	$-\tau^2\alpha_{73} + 1$
63	$5\tau + 1$	(1, 0, 0, 0, 0, 0, -1)	$\tau^2\alpha_{17} + \alpha_{91}$
65	$5\tau + 3$	(1, 0, 0, 0, 0, 0, 1)	$\tau^2\alpha_{17} + \alpha_{93}$
67	$2\tau - 9$	(-1, 0, 0, 1, 0, -1)	$-\tau^2\alpha_{87} - 1$
69	$2\tau - 7$	(-1, 0, 0, 1, 0, 1)	$-\tau^2\alpha_{87} + 1$
71	$2\tau - 5$	(-1, 0, -1, 0, 0, -1)	$-\tau^3\alpha_{91} - 1$
73	$2\tau - 3$	(-1, 0, -1, 0, 0, 1)	$-\tau^3\alpha_{91} + 1$
75	$2\tau - 1$	(-1, 0, -1, 0, -1)	$\tau^2\alpha_{91} - 1$
77	$2\tau + 1$	(-1, 0, -1, 0, 1)	$\tau^2\alpha_{91} + 1$
79	$2\tau + 3$	(1, 0, 1, 0, 0, 0, -1)	$-\tau^2\alpha_{77} - \alpha_{93}$
81	$2\tau + 5$	(1, 0, 1, 0, 0, 0, 1)	$-\tau^2\alpha_{77} - \alpha_{91}$
83	$-\tau - 7$	(-1, 0, -1, 0, 1, 0, -1)	$\tau^2\alpha_{77} - 1$
85	$-\tau - 5$	(-1, 0, -1, 0, 1, 0, 1)	$\tau^2\alpha_{77} + 1$
87	$-\tau - 3$	(1, 0, 0, -1)	$\tau^3 - 1$
89	$-\tau - 1$	(1, 0, 0, 1)	$\tau^3 + 1$
91	$-\tau + 1$	(-1, 0, -1)	$-\tau^2 - 1$
93	$-\tau + 3$	(-1, 0, 1)	$-\tau^2 + 1$
95	$-\tau + 5$	(1, 0, 0, 0, 0, -1)	$\tau^2\alpha_{89} + \alpha_{91}$
97	$-\tau + 7$	(1, 0, 0, 0, 0, 1)	$\tau^2\alpha_{89} + \alpha_{93}$
99	$-\tau + 9$	(-1, 0, -1, 0, 0, 0, 1, 0, -1)	$-\tau^2\alpha_{79} - 1$
101	$-4\tau - 3$	(-1, 0, 0, 0, 1, 0, 1)	$-\tau^2\alpha_{15} + 1$
103	$-4\tau - 1$	(-1, 0, 0, -1, 0, 0, -1)	$-\tau^3\alpha_{89} - 1$
105	$-4\tau + 1$	(-1, 0, 0, -1, 0, 0, 1)	$-\tau^3\alpha_{89} + 1$
107	$-4\tau + 3$	(1, 0, -1, 0, -1)	$-\tau^2\alpha_{93} - 1$
109	$-4\tau + 5$	(1, 0, -1, 0, 1)	$-\tau^2\alpha_{93} + 1$
111	$-4\tau + 7$	(-1, 0, 0, -1, 0, 0, 0, -1)	$-\tau^2\alpha_3 + \alpha_{93}$
113	$-4\tau + 9$	(-1, 0, 0, -1, 0, 0, 0, 1)	$-\tau^2\alpha_3 + \alpha_{91}$
115	$-4\tau + 11$	(-1, 0, -1, 0, 1, 0, 1, 0, -1)	$\tau^2\alpha_{85} - 1$
117	$-7\tau - 1$	(-1, 0, 1, 0, 1, 0, 1)	$-\tau^2\alpha_{107} + 1$
119	$-7\tau + 1$	(-1, 0, -1, 0, 1, 0, 0, -1)	$\tau^3\alpha_{77} - 1$
121	$-7\tau + 3$	(-1, 0, -1, 0, 1, 0, 0, 1)	$\tau^3\alpha_{77} + 1$
123	$-7\tau + 5$	(-1, 0, -1, 0, 0, -1, 0, -1)	$\tau^2\alpha_{71} - 1$
125	$-7\tau + 7$	(-1, 0, -1, 0, 0, -1, 0, 1)	$\tau^2\alpha_{71} + 1$
127	$-7\tau + 9$	(-1, 0, 0, 0, 0, 0, 0, -1)	$-\tau^2\alpha_{95} + \alpha_{91}$

$a = 1$.