Software Implementation of Koblitz Curves over Quadratic Fields

Thomaz Oliveira[1(✉)], Julio López[2], and Francisco Rodríguez-Henríquez[1]

[1] Computer Science Department, CINVESTAV-IPN, Mexico, Mexico
thomaz.figueiredo@gmail.com, jlopez@ic.unicamp.br
[2] Institute of Computing, University of Campinas, Campinas, Brazil
francisco@cs.cinvestav.mx

Abstract. In this work, we retake an old idea presented by Koblitz in his landmark paper [21], where he suggested the possibility of defining anomalous elliptic curves over the base field $\mathbb{F}_4$. We present a careful implementation of the base and quadratic field arithmetic required for computing the scalar multiplication operation in such curves. In order to achieve a fast reduction procedure, we adopted a redundant trinomial strategy that embeds elements of the field $\mathbb{F}_{4^m}$, with m a prime number, into a ring of higher order defined by an almost irreducible trinomial. We also report a number of techniques that allow us to take full advantage of the native vector instructions of high-end microprocessors. Our software library achieves the fastest timings reported for the computation of the timing-protected scalar multiplication on Koblitz curves, and competitive timings with respect to the speed records established recently in the computation of the scalar multiplication over prime fields.

1 Introduction

Anomalous binary curves, generally referred to as Koblitz curves, are binary elliptic curves satisfying the Weierstrass equation, $E_a : y^2 + xy = x^3 + ax^2 + 1$, with $a \in \{0, 1\}$. Since their introduction in 1991 by Koblitz [21], these curves have been extensively studied for their additional structure that allows, in principle, a performance speedup in the computation of the elliptic curve point multiplication operation. As of today, the research works dealing with standardized Koblitz curves in commercial use, such as the binary curves standardized by NIST [23] or the suite of elliptic curves supported by the TLS protocol [4,9], have exclusively analyzed the security and performance of curves defined over binary extension fields $\mathbb{F}_{2^m}$, with m a prime number (for recent examples see [1,5,32,36]).

Nevertheless, Koblitz curves defined over $\mathbb{F}_4$ were also proposed in [21]. We find interesting to explore the cryptographic usage of Koblitz curves defined over

T. Oliveira, F. Henríquez—The authors would like to thank CONACyT (project number 180421) for their funding of this research.

J. López—The author was supported in part by the Intel Labs University Research Office and by a research productivity scholarship from CNPq Brazil.

© International Association for Cryptologic Research 2016
B. Gierlichs and A.Y. Poschmann (Eds.): CHES 2016, LNCS 9813, pp. 259–279, 2016.
DOI: 10.1007/978-3-662-53140-2_13

$\mathbb{F}_4$ due to their inherent usage of quadratic field arithmetic. Indeed, it has been recently shown [3,25] that quadratic field arithmetic is extraordinarily efficient when implemented in software. This is because one can take full advantage of the Single Instruction Multiple Data (SIMD) paradigm, where a vector instruction performs simultaneously the same operation on a set of input data items.

Quadratic extensions of a binary finite field $\mathbb{F}_{q^2}$ can be defined by means of a monic polynomial $h(u)$ of degree two irreducible over $\mathbb{F}_q$. The field $\mathbb{F}_{q^2}$ is isomorphic to $\mathbb{F}_q[u]/(h(u))$ and its elements can be represented as $a_0 + a_1 u$, with $a_0, a_1 \in \mathbb{F}_q$. The addition of two elements $a, b \in \mathbb{F}_{q^2}$, can be performed as $c = (a_0 + b_0) + (a_1 + b_1)u$. By choosing $h(u) = u^2 + u + 1$, the multiplication of a, b can be computed as, $d = a_0 b_0 + a_1 b_1 + ((a_0 + a_1) \cdot (b_0 + b_1) + a_0 b_0)u$. By carefully organizing the code associated to these arithmetic operations, one can greatly exploit the pipelines and their inherent instruction-level parallelism that are available in contemporary high-end processors.

Our Contributions. In this work we designed for the first time, a 128-bit secure and timing attack resistant scalar multiplication on a Koblitz curve defined over $\mathbb{F}_4$, as they were proposed by Koblitz in his 1991 seminal paper [21]. We developed all the required algorithms for performing such a computation. This took us to reconsider the strategy of using redundant trinomials (also known as almost irreducible trinomials), which were proposed more than ten years ago in [6,10]. We also report what is perhaps the most comprehensive analysis yet reported of how to efficiently implement arithmetic operations in binary finite fields and their quadratic extensions using the vectorized instructions available in high-end microprocessors. For example, to the best of our knowledge, we report for the first time a 128-bit AVX implementation of the linear pass technique, which is useful against side-channel attacks.

The remaining of this paper is organized as follows. In Sect. 2 we formally introduce the family of Koblitz elliptic curves defined over $\mathbb{F}_4$. In Sects. 3 and 4 a detailed description of the efficient implementation of the base and quadratic field arithmetic using vectorized instructions is given. We present in Sect. 5 the scalar multiplication algorithms used in this work, and we present in Sect. 6 the analysis and discussion of the results obtained by our software library. Finally, we draw our concluding remarks and future work in Sect. 7.

2 Koblitz Curves over $\mathbb{F}_4$

Koblitz curves over $\mathbb{F}_4$ are defined by the following equation

$$E_a : y^2 + xy = x^3 + a\gamma x^2 + \gamma, \tag{1}$$

where $\gamma \in \mathbb{F}_{2^2}$ satisfies $\gamma^2 = \gamma + 1$ and $a \in \{0, 1\}$. Note that the number of points in the curves $E_0(\mathbb{F}_4)$ and $E_1(\mathbb{F}_4)$ are, $\#E_0(\mathbb{F}_4) = 4$ and $\#E_1(\mathbb{F}_4) = 6$, respectively. For cryptographic purposes, one uses Eq. (1) operating over extension fields of the form $\mathbb{F}_q$, with $q = 4^m$, and m a prime number. The set of affine

points $P = (x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ that satisfy Eq. (1) together with a point at infinity represented as $\mathcal{O}$, forms an abelian group denoted by $E_a(\mathbb{F}_{4^m})$, where its group law is defined by the point addition operation.

Since for each proper divisor l of k, $E(\mathbb{F}_{4^l})$ is a subgroup of $E(\mathbb{F}_{4^k})$, one has that $\#E(\mathbb{F}_{4^l})$ divides $\#E(\mathbb{F}_{4^k})$. Furthermore, by choosing prime extensions m, it is possible to find $E_a(\mathbb{F}_{4^m})$ with almost-prime order, for instance, $E_0(\mathbb{F}_{2^{2\cdot 163}})$ and $E_1(\mathbb{F}_{2^{2\cdot 167}})$. In the remaining of this paper, we will show that the aforementioned strategy can be used for the efficient implementation of a 128-bit secure scalar multiplication on software platforms counting with 64-bit carry-less native multipliers, such as the ones available in contemporary personal desktops.

The Frobenius map $\tau : E_a(\mathbb{F}_q) \rightarrow E_a(\mathbb{F}_q)$ defined by $\tau(\mathcal{O}) = \mathcal{O}$, $\tau(x, y) = (x^4, y^4)$, is a curve automorphism satisfying $(\tau^2 + 4)P = \mu\tau(P)$ for $\mu = (-1)^a$ and all $P \in E_a(\mathbb{F}_q)$. By solving the equation $\tau^2 + 4 = \mu\tau$, the Frobenius map can be seen as the complex number $\tau = (\mu \pm \sqrt{-15})/2$.

2.1 The τ-adic Representation

Given a Koblitz curve $E_a/\mathbb{F}_{2^{2m}}$ with group order $\#E_a(\mathbb{F}_{2^{2m}}) = h \cdot \rho \cdot r$, where h is the order $\#E_a(\mathbb{F}_4)$, r is the prime order of our subgroup of interest, and ρ is the order of a group of no cryptographic interest.[1] We can express a scalar $k \in \mathbb{Z}_r$ as an element in $\mathbb{Z}[\tau]$ using the now classical partial reduction introduced by Solinas [31], with a few modifications. The modified version is based on the fact that $\tau^2 = \mu\tau - 4$.

Given that the norm of τ is $N(\tau) = 4$, $N(\tau - 1) = h$, $N(\tau^m - 1) = h \cdot \rho \cdot r$ and $N((\tau^m - 1)/(\tau - 1)) = \rho \cdot r$, the subscalars r_0 and r_1 resulting from the partial modulo function will be both of size approximately $\sqrt{\rho \cdot r}$. As a consequence, the corresponding scalar multiplication will need more iterations than expected, since it will consider the order ρ of a subgroup which is not of cryptographic interest.

For that reason, we took the design decision of considering that the input scalar of our point multiplication algorithm is already given in the $\mathbb{Z}[\tau]$ domain. As a result, a partial reduction of the scalar k is no longer required, and the number of iterations in the point multiplication will be consistent with the scalar k size. If one needs to retrieve the equivalent value of the scalar k in the ring $\mathbb{Z}_r$, this can be easily computed with one multiplication and one addition in $\mathbb{Z}_r$. This strategy is in line with the degree-2 scalar decomposition method within the GLS curves context as suggested in [12].

2.2 The Width-w τNAF Form

Assuming that the scalar k is specified in the $\mathbb{Z}[\tau]$ domain, one can represent the scalar in the regular width-w τNAF form as shown in Algorithm 1. The length of the representation width-w τNAF of an element $k \in \mathbb{Z}[\tau]$ is discussed in [30].

[1] Usually the order ρ is composite. Also, every prime factor of ρ is smaller than r.

Algorithm 1. Regular width-w τ-recoding for m-bit scalar

Input: w, t_w, $\alpha_v = \beta_v + \gamma_v\tau$ for $v = \{\pm 1, \pm 3, \pm 5, \ldots, \pm 4^{w-1} - 1\}$, $\rho = r_0 + r_1\tau \in \mathbb{Z}[\tau]$ with odd r_0, r_1

Output: $\rho = \sum_{i=0}^{\lceil \frac{m+2}{w-1} \rceil} v_i \tau^{i(w-1)}$

1: **for** $i \leftarrow 0$ **to** $\lceil \frac{m+2}{w-1} \rceil$ - 1 **do**
2: **if** $w = 2$ **then**
3: $v_i \leftarrow ((r_0 - 4 \cdot r_1) \bmod 8) - 4$
4: $r_0 \leftarrow r_0 - v_i$
5: **else**
6: $u \leftarrow (r_0 + r_1 t_w \bmod 2^{2w-1}) - 2^{2(w-1)}$
7: **if** $v > 0$ **then** $s \leftarrow 1$ **else** $s \leftarrow -1$
8: $r_0 \leftarrow r_0 - s\beta_v, r_1 \leftarrow r_1 - s\gamma_v, v_i \leftarrow s\alpha_v$
9: **end if**
10: **for** $j \leftarrow 0$ **to** $(w - 2)$ **do**
11: $t \leftarrow r_0, r_0 \leftarrow r_1 + (\mu \cdot r_0)/4, r_1 \leftarrow -t/4$
12: **end for**
13: **end for**
14: **if** $r_0 \neq 0$ **and** $r_1 \neq 1$ **then**
15: $v_i \leftarrow r_0 + r_1\tau$
16: **else**
17: **if** $r_1 \neq 0$ **then**
18: $v_i \leftarrow r_1$
19: **else**
20: $v_i \leftarrow r_0$
21: **end if**
22: **end if**

Given a width w, after running Algorithm 1, we have $2^{2(w-1)-1}$ different digits.[2] As a result, it is necessary to be more conservative when choosing the width w, when compared to the Koblitz curves defined over $\mathbb{F}_2$. For widths $w = 2, 3, 4, 5$ we have to pre- or post-compute $2, 8, 32$ and 128 points, respectively. For the 128-bit point multiplication, we estimated that the value of the width w must be at most four, otherwise, the costs of the point pre/post-processing are greater than the addition savings obtained in the main iteration.

In addition, we must find efficient expressions of $\alpha_v = v \bmod \tau^w$. The method for searching the best expressions in Koblitz curves over $\mathbb{F}_2$ [33] cannot be directly applied in the $\mathbb{F}_4$ case. Because of this, we manually provided α_v representations for $w \in \{2, 3, 4\}$ and $a = 1$, which are our implementation parameters. The main rationale for our representation choices was to minimize the number of field arithmetic operations. In practice, we strive for reducing the required number of full point additions by increasing the number of point doublings and mixed additions, which have a cheaper computational cost.[3] In Table 1 we present the α_v representatives along with the operations required to generate the multiples of the base point.[4]

Therefore, one point doubling and one full addition are required to generate the points $\alpha_v \cdot P$ for $w = 2$, one point doubling, four full additions, three

[2] We are considering only positive digits, since the cost of computing the negative points in binary elliptic curves is negligible.

[3] Full addition is defined as the addition of two points given in projective coordinates. The mixed addition operation adds one point given in projective coordinates with another given in affine coordinates.

[4] Notice that the multiples $\alpha_v \cdot P$ as shown in Table 1, must be computed out of order. The order for computing the multiples is shown in roman numbers.

Table 1. Representations of $\alpha_v = v \bmod \tau^w$, for $w \in \{2,3,4\}$ and $a = 1$ and the required operations for computing α_v. Here we denote by D, FA, MA, T the point doubling, full addition, mixed addition and the Frobenius map, respectively. In addition, we consider that the point $\alpha_1 P$ is represented in affine coordinates. The order for computing the points is given in roman numbers

w	v	$v \bmod \tau^w$	α_v	Operations	Order
2	1	1	1	n/a	I
	3	3	3	$t_0 \leftarrow 2\alpha_1$, $\alpha_3 \leftarrow t_0 + \alpha_1$ $(D + FA)$	II
3	1	1	1	n/a	I
	3	3	3	$t_0 \leftarrow 2\alpha_1$, $\alpha_3 \leftarrow t_0 + \alpha_1$ $(D + FA)$	II
	5	5	$-\tau - \alpha_{15}$	$\alpha_5 \leftarrow -t_1 - \alpha_{15}$ (MA)	VIII
	7	$3\tau + 3$	$\tau^2\alpha_3 + \alpha_3$	$\alpha_7 \leftarrow \tau^2\alpha_3 + \alpha_3$ $(FA + 2T)$	III
	9	$3\tau + 5$	$\alpha_7 + 2$	$\alpha_9 \leftarrow \alpha_7 + t_0$ (FA)	IV
	11	$3\tau + 7$	$\alpha_9 + 2$	$\alpha_{11} \leftarrow \alpha_9 + t_0$ (FA)	V
	13	$-\tau - 7$	$\tau^2 - \alpha_3$	$\alpha_{13} \leftarrow t_2 - \alpha_3$ (MA)	VII
	15	$-\tau - 5$	$\tau^2 - 1$	$t_1 \leftarrow \tau\alpha_1$, $t_2 \leftarrow \tau t_1$, $\alpha_{15} \leftarrow t_2 - \alpha_1$ $(MA + 2T)$	VI
4	1	1	1	n/a	I
	3	3	$-\tau^3 - \alpha_{61}$	$\alpha_3 \leftarrow -t_4 - \alpha_{61}$ (MA)	XXVI
	5	5	$-\tau^3 - \alpha_{59}$	$\alpha_5 \leftarrow -t_4 - \alpha_{59}$ (MA)	XXVII
	7	7	$-\tau^3 - \alpha_{57}$	$\alpha_7 \leftarrow -t_4 - \alpha_{57}$ (MA)	XXVIII
	9	9	$-\tau^3 - \alpha_{55}$	$\alpha_9 \leftarrow -t_4 - \alpha_{55}$ (MA)	XXIX
	11	11	$-2\tau^2 + \alpha_{43}$	$\alpha_{11} \leftarrow -t_2 + \alpha_{43}$ (FA)	XXX
	13	13	$-2\tau^2 + \alpha_{45}$	$\alpha_{13} \leftarrow -t_2 + \alpha_{45}$ (FA)	XXXI
	15	15	$-2\tau^2 + \alpha_{47}$	$\alpha_{15} \leftarrow -t_2 + \alpha_{47}$ (FA)	XXXII
	17	$5\tau - 11$	$-\tau^3 - \alpha_{47}$	$t_4 \leftarrow \tau^2 t_3$, $\alpha_{17} \leftarrow -t_4 - \alpha_{47}$ $(MA + 2T)$	XIX
	19	$5\tau - 9$	$-\tau^3 - \alpha_{45}$	$\alpha_{17} \leftarrow -t_4 - \alpha_{47}$ (MA)	XX
	21	$5\tau - 7$	$-\tau^3 - \alpha_{43}$	$\alpha_{17} \leftarrow -t_4 - \alpha_{45}$ (MA)	XXI
	23	$5\tau - 5$	$-\tau^3 - \alpha_{41}$	$\alpha_{17} \leftarrow -t_4 - \alpha_{43}$ (MA)	XXII
	25	$5\tau - 3$	$-\tau^3 - \alpha_{39}$	$\alpha_{17} \leftarrow -t_4 - \alpha_{41}$ (MA)	XXIII
	27	$5\tau - 1$	$-\tau^3 - \alpha_{37}$	$\alpha_{17} \leftarrow -t_4 - \alpha_{39}$ (MA)	XXIV
	29	$5\tau + 1$	$-\tau^3 - \alpha_{35}$	$\alpha_{17} \leftarrow -t_4 - \alpha_{37}$ (MA)	XXV
	31	$-2\tau - 9$	$2\tau^2 - 1$	$t_2 \leftarrow \tau t_1$, $\alpha_{31} \leftarrow t_2 - \alpha_1$ $(MA + T)$	XII
	33	$-2\tau - 7$	$2\tau^2 + 1$	$\alpha_{33} \leftarrow t_2 + \alpha_1$ (MA)	XIII
	35	$-2\tau - 5$	$-2\tau - 5$	$\alpha_{35} \leftarrow \alpha_{37} - t_0$ (FA)	VI
	37	$-2\tau - 3$	$-2\tau - 3$	$\alpha_{37} \leftarrow \alpha_{39} - t_0$ (FA)	IV
	39	$-2\tau - 1$	$-2\tau - 1$	$t_0 \leftarrow 2\alpha_1$, $t_1 \leftarrow \tau t_0$, $\alpha_{39} \leftarrow -t_1 - \alpha_1$ $(D + MA + T)$	II
	41	$-2\tau + 1$	$-2\tau + 1$	$\alpha_{41} \leftarrow -t_1 + \alpha_1$ (MA)	III
	43	$-2\tau + 3$	$-2\tau + 3$	$\alpha_{43} \leftarrow \alpha_{41} + t_0$ (FA)	V
	45	$-2\tau + 5$	$-2\tau + 5$	$\alpha_{45} \leftarrow \alpha_{43} + t_0$ (FA)	VII
	47	$-2\tau + 7$	$-2\tau + 7$	$\alpha_{47} \leftarrow \alpha_{45} + t_0$ (FA)	VIII
	49	$-2\tau + 9$	$-2\tau + 9$	$\alpha_{49} \leftarrow \alpha_{47} + t_0$ (FA)	IX
	51	$-2\tau + 11$	$-2\tau + 11$	$\alpha_{51} \leftarrow \alpha_{49} + t_0$ (FA)	X
	53	$-2\tau + 13$	$-2\tau + 13$	$\alpha_{53} \leftarrow \alpha_{51} + t_0$ (FA)	XI
	55	$3\tau - 13$	$3\tau - 13$	$t_3 \leftarrow \tau\alpha_1$, $\alpha_{55} \leftarrow t_3 - \alpha_{53}$ $(MA + T)$	XIV
	57	$3\tau - 11$	$3\tau - 11$	$\alpha_{57} \leftarrow t_3 - \alpha_{51}$ (MA)	XV
	59	$3\tau - 9$	$3\tau - 9$	$\alpha_{59} \leftarrow t_3 - \alpha_{49}$ (MA)	XVI
	61	$3\tau - 7$	$3\tau - 7$	$\alpha_{61} \leftarrow t_3 - \alpha_{47}$ (MA)	XVII
	63	$3\tau - 5$	$3\tau - 5$	$\alpha_{63} \leftarrow t_3 - \alpha_{45}$ (MA)	XVIII

mixed additions and four applications of the Frobenius map for the $w = 3$ case and one point doubling, twenty full additions, eleven mixed additions and five applications of the Frobenius map for the $w = 4$ case.

2.3 Security of the Koblitz Curves Defined over $\mathbb{F}_4$

Since the Koblitz curves defined over $E_a(\mathbb{F}_{4^m})$ operate over quadratic extensions fields, it is conceivable that Weil descent attacks [13,16] could possibly be efficiently applied on these curves. However, Menezes and Qu showed in [22] that the GHS attack cannot be implemented efficiently for elliptic curves defined over binary extension fields $\mathbb{F}_q$, with $q = 2^m$, and m a prime number in $[160, \ldots, 600]$. Further, a specialized analysis for binary curves defined over fields of the form $\mathbb{F}_{4^m}$ reported in [14], proved that the only vulnerable prime extension in the range $[80, \ldots, 256]$, is $m = 127$. Therefore, the prime extension used in this work, namely, $m = 149$, is considered safe with respect to the state-of-the-art knowledge of the Weil descent attack classes.

For a comprehensive survey of recent progress in the computation of the elliptic curve discrete problem in characteristic two, the reader is referred to the paper by Galbraith and Gaudry [11].

3 Base Field Arithmetic

In this section, we present the techniques used in our work in order to implement the binary field arithmetic. We selected a Koblitz curve with the parameter $a = 1$ defined over $\mathbb{F}_{4^m}$ with $m = 149$. This curve was chosen because the order of its subgroup of interest is of size 2^{254}, which yields a security level roughly equivalent to a 128-bit secure scalar multiplication.

3.1 Modular Reduction

One can construct a binary extension field $\mathbb{F}_{2^m}$ by taking a polynomial $f(x) \in \mathbb{F}_2[x]$ of degree m which is irreducible over $\mathbb{F}_2$. It is very important that the form of the polynomial $f(x)$, admits an efficient modular reduction. The criteria for selecting $f(x)$ depends on the architecture where the implementation will be executed as it was extensively discussed in [29].

For our field extension choice, we do not have degree-149 trinomials which are irreducible over $\mathbb{F}_2$. An alternative solution is to construct the field through irreducible pentanomials. Given an irreducible pentanomial $f(x) = x^m + x^a + x^b + x^c + 1$, the efficiency of the shift-and-add reduction method depends mostly on the fact that the term-degree differences $m - a$, $m - b$ and $m - c$, are all equal to 0 modulo W, where W is the architecture word size in bits.

Using the terminology of [29], lucky irreducible pentanomials are the ones where the three previously mentioned differences are equal to 0 modulo W. Fortunate irreducible pentanomials are the ones where two out of the three above differences are equal to 0 modulo W. The remaining cases are called

ordinary irreducible pentanomials. Performing an extensive search with $W = 8$, we found no lucky pentanomials, 189 fortunate pentanomials and 9491 ordinary pentanomials for the extension $m = 149$.

The problem is that fortunate pentanomials make the modular reduction too costly if we compare it with the field multiplication computed with carry-less instructions. This is because we need to perform four shift-and-add operations per reduction step. Besides, two of those operations require costly shift instructions, since they are shifts not divisible by 8.

3.2 Redundant Trinomials

As a consequence of the above analysis, we resorted to the redundant trinomials strategy introduced in [6,10], also known as almost irreducible trinomials. Given a non-irreducible trinomial $g(x)$ of degree n that factorizes into an irreducible polynomial $f(x)$ of degree $m < n$, the idea is to perform the field reduction modulo $g(x)$ throughout the scalar multiplication and, at the end of the algorithm, reduce the polynomials so obtained modulo $f(x)$. In a nutshell, throughout the algorithm we represent the base field elements as polynomials in the ring $\mathbb{F}_2[x]$ reduced modulo $g(x)$. At the end of the algorithm, the elements are reduced modulo $f(x)$ in order to bring them back to the target field $\mathbb{F}_{2^{149}}$. For the sake of simplicity, throughout this paper, we will refer to those elements as field elements.

Since our target software platform counts with a native 64-bit carry-less multiplier, an efficient representation of the field elements must have at most 192 bits, *i.e*, three 64-bit words. For that reason, we searched for redundant trinomials of degree at most 192.

We selected the trinomial, $g(x) = x^{192} + x^{19} + 1$, for two reasons. First, since our target architecture contains 128-bit vectorized registers, the difference $(m - a) > 128$ allows us to perform the shift-and-add reduction in just two steps. Second, the property $m \bmod 64 = 0$, which allows us to perform efficiently the first part of the shift-and-add reduction. The steps to perform the modular reduction are described in Algorithm 2.[5] The reduction using 128-bit registers is presented in Sect. 4, where we discuss our strategy for implementing the arithmetic in the quadratic field extension.

Algorithm 2. Modular reduction by the trinomial $g(x) = x^{192} + x^{19} + 1$

Input: A 384-bit polynomial $r(x) = F \cdot x^{320} + E \cdot x^{256} + D \cdot x^{192} + C \cdot x^{128} + B \cdot x^{64} + A$ in $\mathbb{F}_2[x]$ stored into six 64-bit registers (A - F).

Output: A 192-bit polynomial $s(x) = r(x) \bmod g(x) = I \cdot x^{128} + H \cdot x^{64} + G$ stored into three 64-bit registers (G - I).

1: $G \leftarrow A \oplus D \oplus (F \gg 45) \oplus ((D \oplus (F \gg 45)) \ll 19)$
2: $H \leftarrow B \oplus E \oplus (E \ll 19) \oplus (D \gg 45)$
3: $I \leftarrow C \oplus F \oplus (F \ll 19) \oplus (E \gg 45)$

[5] The symbols $\ll, \gg$ stand for bitwise shift of packed 64-bit integers.

The overall cost of the modular reduction is ten `xors` and five bitwise shifts. At the end of the scalar multiplication, we have to reduce the 192-bit polynomial to an element of the field $\mathbb{F}_{2^{149}}$. Note that the trinomial $g(x) = x^{192} + x^{19} + 1$ factorizes into a 69-term irreducible polynomial $f(x)$ of degree 149.

The final reduction is performed via the mul-and-add reduction which, experimentally, performed more efficiently than the shift-and-add reduction.[6] Concisely, the mul-and-add technique consists in a series of steps which includes shift operations (in order to align the bits in the registers), carry-less multiplications and `xor` operations for eliminating the extra bits.

The basic mul-and-add step is described in Algorithm 3. Here, besides the usual notation, we represent the 64-bit carry-less multiplication by the symbol $\times_{ij}$, where $i, j = \{L, H\}$, with L and H representing the lowest and highest 64-bit word packed in a 128-bit register, respectively. For example, if one wants to multiply the 128-bit register A lowest 64-bit word by the 128-bit register B highest 64-bit word, we would express this operation as $T \leftarrow A \times_{LH} B$.

Algorithm 3. Basic step of the mul-and-add reduction modulo the 69-term irreducible polynomial $f(x)$

Input: A j-bit polynomial $r(x) = B \cdot x^{128} + A$ stored into two 128-bit registers (A, B), for $j \in [191, 148]$, the irreducible polynomial $f(x) = F \cdot x^{128} + E$ stored into two 128-bit registers (E, F).

Output: A $(j-3)$-bit polynomial $s(x) = D \cdot x^{128} + C$ stored into two 128-bit registers (C, D).

1: $T_0 \leftarrow B \gg 21$ (64-bit alignment)
2: $T_1 \leftarrow E \times_{LL} T_0$
3: $T_2 \leftarrow E \times_{HL} T_0$
4: $T_0 \leftarrow F \times_{LL} T_0$
5: $T_1 \leftarrow T_1 \oplus (T_2 \ll 64)$
6: $T_0 \leftarrow T_0 \oplus (T_2 \gg 64)$
7: $C \leftarrow A \oplus T_1$
8: $D \leftarrow B \oplus T_0$

Algorithm 3 requires four `xors`, three bitwise shifts and three carry-less multiplications. In our particular case, the difference between the degrees of the two most significant monomials of $f(x)$ is three. Also, note that we need to reduce 43 bits (191–148). As a result, it is required $\lceil \frac{43}{3} \rceil = 15$ applications of the Algorithm 3 in order to conclude this reduction.

4 Quadratic Field Arithmetic

In this Section, the basic arithmetic operations in the quadratic field are presented. As usual, the quadratic field $\mathbb{F}_{2^{2 \cdot 149}}$ was constructed by the degree two monic polynomial $h(u) = u^2 + u + 1$, and its elements are represented as $a_0 + a_1 u$, with $a_0, a_1 \in \mathbb{F}_{2^{149}}$.

[6] For a more detailed explanation of the shift-and-add and the mul-and-add reduction methods to binary fields, see [5].

4.1 Register Allocation

The first aspect to be considered is the element allocation into the architecture's available registers. In our case, we have to store two polynomials of 192 bits into 128-bit registers in such a way that it allows an efficient modular reduction and, at the same time, it generates a minimum overhead in the two main arithmetic operations, namely, the multiplication and squaring.

Let us consider an element $a = (a_0 + a_1 u) \in \mathbb{F}_{2^{2 \cdot 149}}$, where $a_0 = C \cdot x^{128} + B \cdot x^{64} + A$ and $a_1 = F \cdot x^{128} + E \cdot x^{64} + D$ are 192-bit polynomials, each one of them stored into three 64-bit words (A-C, D-F). Also, let us have three 128-bit registers R_i, with $i \in \{0, 1, 2\}$, which can store two 64-bit words each. In this Section, we adopted the following notation, given a 128-bit register R, its most and least significant packed 64-bit words, denoted respectively by S and T, are represented as $R = S|T$. The first option is to rearrange the 384-bit element $a = (a_0 + a_1 u)$ as,

$$R_0 = A|B, \quad R_1 = C|D, \quad R_2 = E|F.$$

The problem with this representation is that a significant overhead is generated in the multiplication function, more specifically, in the pre-computation phase of the Karatsuba procedure (cf. Sect. 4.2 with the computation of $V_{0,1}$, $V_{0,2}$ and $V_{1,2}$). Besides, in order to efficiently perform the subsequent reduction phase, we must temporarily store the polynomial terms into four 128-bit vectors, which can cause a register overflow. A better method for storing the element a is to use the following arrangement,

$$R_0 = D|A, \quad R_1 = E|B, \quad R_2 = F|C.$$

Using this setting, there still exists some overhead in the multiplication and squaring arithmetic operations, even though the penalty on the latter operation is almost negligible. In the positive side, the terms of the elements a_0, a_1 do not need to be rearranged and the modular reduction of these two base field elements can be performed in parallel, as discussed next.

4.2 Multiplication

Given two $\mathbb{F}_{2^{2 \cdot 149}}$ elements $a = (a_0 + a_1 u)$ and $b = (b_0 + b_1 u)$, with a_0, a_1, b_0, b_1 in $\mathbb{F}_{2^{149}}$, we perform the multiplication $c = a \cdot b$ as,

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

where each element $a_i, b_i \in \mathbb{F}_{2^{149}}$ is composed by three 64-bit words. The analysis of the Karatsuba algorithm cost for different word sizes was presented in [35]. There, it was shown that the most efficient way to multiply three 64-bit word polynomials $s(x) = s_2 x^2 + s_1 x + s_0$ and $t(x) = t_2 x^2 + t_1 x + t_0$ as $v(x) = s(x) \cdot t(x)$ is through the one-level Karatsuba method,

$$V_0 = s_0 \cdot t_0, \quad V_1 = s_1 \cdot t_1, \quad V_2 = s_2 \cdot t_2,$$

$$V_{0,1} = (s_0 \oplus s_1) \cdot (t_0 \oplus t_1), \quad V_{0,2} = (s_0 \oplus s_2) \cdot (t_0 \oplus t_2) \quad V_{1,2} = (s_1 \oplus s_2) \cdot (t_1 \oplus t_2),$$

$$v(x) = V_2 \cdot x^4 + (V_{1,2} \oplus V_1 \oplus V_2) \cdot x^3 + (V_{0,2} \oplus V_0 \oplus V_1 \oplus V_2) \cdot x^2 + (V_{0,1} \oplus V_0 \oplus V_1) \cdot x + V_0,$$

which costs six multiplications and twelve additions. The Karatsuba algorithm as used in this work is presented in Algorithm 4.[7]

Algorithm 4. Karatsuba algorithm for multiplying three 64-bit word polynomials $s(x)$ and $t(x)$

Input: Six 128-bit registers R_i, with $i \in \{0 \dots 5\}$, containing the elements $R_0 = t_0|s_0, R_1 = t_1|s_1$, $R_2 = t_2|s_2, R_3 = (t_0 \oplus t_1)|(s_0 \oplus s_1)$, $R_4 = (t_0 \oplus t_2)|(s_0 \oplus s_2)$, $R_5 = (t_1 \oplus t_2)|(s_1 \oplus s_2)$.

Output: Three 128-bit registers R_i, with $i \in \{6 \dots 8\}$, which store the value $v(x) = s(x) \cdot t(x) = v_5 \cdot x^{320} + v_4 \cdot x^{256} + v_3 \cdot x^{192} + v_2 \cdot x^{128} + v_1 \cdot x^{64} + v_0$ as $R_6 = v_1|v_0, R_7 = v_3|v_2, R_8 = v_5|v_4$.

1: $tmp_0 \leftarrow R_0 \times_{HL} R_0$
2: $tmp_1 \leftarrow R_1 \times_{HL} R_1$
3: $tmp_2 \leftarrow R_2 \times_{HL} R_2$
4: $tmp_3 \leftarrow R_3 \times_{HL} R_3$
5: $tmp_4 \leftarrow R_4 \times_{HL} R_4$
6: $tmp_5 \leftarrow R_5 \times_{HL} R_5$
7: $tmp_5 \leftarrow tmp_5 \oplus tmp_1$
8: $tmp_5 \leftarrow tmp_5 \oplus tmp_2$
9: $tmp_1 \leftarrow tmp_1 \oplus tmp_0$
10: $tmp_4 \leftarrow tmp_4 \oplus tmp_1$
11: $tmp_4 \leftarrow tmp_4 \oplus tmp_2$
12: $tmp_3 \leftarrow tmp_3 \oplus tmp_1$
13: $R_6 \leftarrow (tmp_3 \ll 64)$
14: $R_8 \leftarrow (tmp_5 \gg 64)$
15: $R_7 \leftarrow ((tmp_5, tmp_3) \rhd 64)$

Algorithm 4 requires six carry-less instructions, six vectorized `xors` and three bitwise shift instructions. In order to calculate the total multiplication cost, it is necessary to include the Karatsuba pre-computation operations at the base field level (twelve vectorized `xors` and six byte interleaving instructions) and at the quadratic field level (six vectorized `xors`). Also, we must consider the reorganization of the registers in order to proceed with the modular reduction (six vectorized `xors`).

4.3 Modular Reduction

The modular reduction of an element $a = (a_0 + a_1 u)$, where a_0 and a_1 are 384-bit polynomials, takes nine vectorized `xors` and six bitwise shifts. The computational savings of the previously discussed register configuration can be seen when we compare the reduction of quadratic field elements, presented in Algorithm 5 with the modular reduction of the base field elements (see Algorithm 2). The cost of reducing an element in $\mathbb{F}_{2^{149}}$ in 64-bit registers is about the same as the cost of the reduction of an element in $\mathbb{F}_{2^{2 \cdot 149}}$ stored into 128-bit registers. Thus, we achieved a valuable speedup of 100 %.

[7] As before, the symbols $\ll, \gg$ stand for bitwise shift of packed 64-bit integers. The symbol $\rhd$ stands for bytewise multi-precision shift.

Algorithm 5. Modular reduction of the terms a_0, a_1 of an element $a = (a_0 + a_1 u)$ modulo $g(x) = x^{192} + x^{19} + 1$

Input: An element $a = a_0 + a_1 u = (F \cdot x^{320} + E \cdot x^{256} + D \cdot x^{192} + C \cdot x^{128} + B \cdot x^{64} + A) + (L \cdot x^{320} + K \cdot x^{256} + J \cdot x^{192} + I \cdot x^{128} + H \cdot x^{64} + G)u$, with the 64-bit words (A-L) arranged in six 128-bit registers as $R_0 = G|A, R_1 = H|B, R_2 = I|C, R_3 = J|D, R_4 = K|E, R_5 = L|F$

Output: Elements (a_0, a_1) mod $g(x) = M \cdot x^{128} + N \cdot x^{64} + O, P \cdot x^{128} + Q \cdot x^{64} + R$, with the 64-bit words (M-R) organized in three 128-bit registers as $R_6 = R|O, R_7 = Q|N, R_8 = P|M$

1: $R_8 \leftarrow R_2 \oplus R_5$
2: $R_7 \leftarrow R_1 \oplus R_4$
3: $R_8 \leftarrow R_8 \oplus (R_5 \ll 19)$
4: $R_7 \leftarrow R_7 \oplus (R_4 \ll 19)$
5: $R_8 \leftarrow R_8 \oplus (R_4 \gg 45)$
6: $R_7 \leftarrow R_7 \oplus (R_3 \ll 45)$
7: $R_6 \leftarrow R_3 \oplus (R_5 \gg 45)$
8: $R_6 \leftarrow R_6 \oplus (R_6 \ll 19)$
9: $R_6 \leftarrow R_6 \oplus R_0$

4.4 Squaring

Squaring is a very important function in the Koblitz curve point multiplication algorithm, since it is the building block for computing the τ endomorphism. In our implementation, we computed the squaring operation through carry-less multiplication instructions which, experimentally, was an approach less expensive than the bit interleaving method (see [15, Sect. 2.3.4]). The pre-processing phase is straightforward, we just need to rearrange the 32-bit packed words of the 128-bit registers in order to prepare them for the subsequent modular reduction.

The pre- and post-processing phases require three shuffle instructions, three vectorized `xors` and three bitwise shifts. The complete function is described in Algorithm 6. Given 128-bit registers R_i, we depict the SSE 32-bit shuffle operation as $R_0 \leftarrow R_1 \lbag xxxx$. For instance, if we compute $R_0 \leftarrow R_1 \lbag 3210$, it just maintains the 32-bit word order of the register R_1, in other words, it just copies R_1 to R_0. The operation $R_0 \leftarrow R_1 \lbag 2103$ rotates the register R_1 to the left by 32-bits. See [17,18] for more details.

4.5 Inversion

The inversion operation is computed via the Itoh-Tsujii method [19]. Given an element $c \in \mathbb{F}_{2^m}$, we compute $c^{-1} = c^{(2^{m-1}-1)\cdot 2}$ through an addition chain, which in each step computes the terms $(c^{2^i-1})^{2^j} \cdot c^{2^j-1}$ with $0 \leq j \leq i \leq m-1$. For the case $m = 149$, the following chain is used,

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 33 \rightarrow 66 \rightarrow 74 \rightarrow 148.$$

This addition chain is optimal and was found through the procedure described in [7]. Note that although we compute the inversion operation over polynomials in $\mathbb{F}_2[x]$ (reduced modulo $g(x) = x^{192} + x^{19} + 1$), we still have to perform the addition chain with $m = 149$, since we are in fact interested in the embedded $\mathbb{F}_{2^{149}}$ field element.

Algorithm 6. Squaring of an element $a = (a_0 + a_1 u) \in \mathbb{F}_{2^{2 \cdot 149}}$

Input: Element $a = a_0 + a_1 u = (C \cdot x^{128} + B \cdot x^{64} + A) + (F \cdot x^{128} + E \cdot x^{64} + D)u \in \mathbb{F}_{2^{2 \cdot 149}}$, with the 64-bit words (A-F) arranged in three 128-bit registers as $R_0 = D|A, R_1 = E|B, R_2 = F|C$

Output: Element $a^2 = c = c_0 + c_1 u = (I \cdot x^{128} + H \cdot x^{64} + G) + (L \cdot x^{128} + K \cdot x^{64} + J)u \in \mathbb{F}_{2^{2 \cdot 149}}$, where both elements $(c_0, c_1) \in \mathbb{F}_2[x]$ are reduced modulo $x^{192} + x^{19} + 1$. The 64-bit words (G-L) are arranged in three 128-bit registers as $R_3 = J|G, R_4 = H|K, R_5 = I|L$.

1: $tmp_0 \leftarrow R_0 \lll 3120$
2: $tmp_1 \leftarrow R_1 \lll 3120$
3: $tmp_2 \leftarrow R_2 \lll 3120$
4: $aux_0 \leftarrow tmp_0 \times_{LL} tmp_0$
5: $aux_1 \leftarrow tmp_0 \times_{HH} tmp_0$
6: $aux_2 \leftarrow tmp_1 \times_{LL} tmp_1$
7: $aux_3 \leftarrow tmp_1 \times_{HH} tmp_1$
8: $aux_4 \leftarrow tmp_2 \times_{LL} tmp_2$
9: $aux_5 \leftarrow tmp_2 \times_{HH} tmp_2$
10: $R_3, R_4, R_5 \leftarrow \texttt{ModularReduction}(aux_{0\ldots5})$
11: $tmp_0 \leftarrow R_3 \gg 64$
12: $tmp_1 \leftarrow R_4 \gg 64$
13: $tmp_2 \leftarrow R_5 \gg 64$
14: $R_3 \leftarrow R_3 \oplus tmp_0$
15: $R_4 \leftarrow R_4 \oplus tmp_1$
16: $R_5 \leftarrow R_5 \oplus tmp_2$

As previously discussed, in each step of the addition chain, we must calculate an exponentiation c^{2^j} followed by a multiplication, where the value j represents the integers that form the addition chain. Experimentally, we found that when $j \geq 4$, it is cheaper to compute the exponentiation through table look-ups instead of performing consecutive squarings. Our pre-computed tables process four bits per iteration, therefore, it is required $\lceil \frac{192}{4} \rceil = 48$ table queries in order to complete the multisquaring function.

5 τ-and-add Scalar Multiplication

In this Section we discuss the single-core algorithms that compute a timing-resistant scalar multiplication through the τ-and-add method over Koblitz curves defined over $\mathbb{F}_4$. There are two basic approaches, the right-to-left and the left-to-right algorithms.

5.1 Left-to-right τ-and-add

This algorithm is similar to the traditional left-to-right double-and-add method. Here, the point doubling operation is replaced by the computationally cheaper τ endomorphism. In addition, we need to compute the width w-τNAF representation of the scalar k and perform linear passes (cf. Sect. 5.3) in the accumulators in order to avoid cache attacks [26,34]. The method is shown in Algorithm 7.

The main advantage of this method is that the sensitive data is indirectly placed in the points P_{v_i}. However, those points are only read and then added to the unique accumulator Q. As a consequence, only one linear pass per iteration is required before reading P_{v_i}. On the other hand, the operation $\tau^{w-1}(Q)$ must be performed by successive squarings, since computing it through look-up tables could leak information about the scalar k.

Algorithm 7. Left-to-right regular w-TNAF τ-and-add on Koblitz curves defined over $\mathbb{F}_4$

Input: A Koblitz curve $E_a/\mathbb{F}_{2^{2m}}$, a point $P \in E_a(\mathbb{F}_{2^{2m}})$ of order r, $k \in \mathbb{Z}_r$
Output: $Q = kP$

Compute $\rho = r_0 + r_1\tau = k$ partmod $\left(\frac{\tau^m - 1}{\tau - 1}\right)$
Ensure that r_0 and r_1 are odd.
Compute the width-w regular τ-NAF of $r_0 + r_1\tau$ as $\sum_{i=0}^{\lceil \frac{m+2}{w-1} + 1 \rceil} v_i \tau^{i(w-1)}$
for $v \in \{1, 3, \ldots 4^{w-1} - 1\}$ **do** Compute $P_v = \alpha_v \cdot P$ **end for**
$Q \leftarrow \mathcal{O}$
for $i = \frac{m+2}{w-1} + 1$ **to** 0 **do**
    $Q \leftarrow \tau^{w-1}(Q)$
    Perform a linear pass to recover P_{v_i}
    $Q \leftarrow Q \pm P_{v_i}$
end for
Subtract $P, \tau(P)$ from Q if necessary
return $Q = kP$

5.2 Right-to-left τ-and-add

This other method processes the scalar k from the least to the most significant digit. Taking advantage of the τ endomorphism, the GLV method is brought to its full extent. This approach is presented in Algorithm 8.

Algorithm 8. Right-to-left regular w-TNAF τ-and-add on Koblitz curves defined over $\mathbb{F}_4$

Input: A Koblitz curve $E_a/\mathbb{F}_{2^{2m}}$, a point $P \in E_a(\mathbb{F}_{2^{2m}})$ of order r, $k \in \mathbb{Z}_r$
Output: $Q = kP$

Compute $\rho = r_0 + r_1\tau = k$ partmod $\pmod{\frac{\tau^m - 1}{\tau - 1}}$
Ensure that r_0 and r_1 are odd.
Compute the width-w regular τ-NAF of $r_0 + r_1\tau$ as $\sum_{i=0}^{\lceil \frac{m+2}{w-1} + 1 \rceil} v_i \tau^{i(w-1)}$
for $i \in \{1, 3, \ldots 4^{w-1} - 1\}$ **do** $Q_i = \mathcal{O}$
for $i = 0$ **to** $\frac{m+2}{w-1} + 1$ **do**
    Perform a linear pass to recover Q_i
    $Q_i \leftarrow Q_i \pm P$
    Perform a linear pass to store Q_i
    $P \leftarrow \tau^{w-1}(P)$
$Q \leftarrow \mathcal{O}$
for $u \in \{1, 3, \ldots 4^{w-1} - 1\}$ **do** $Q = Q + \alpha_v \cdot Q_i$
Subtract $P, \tau(P)$ from Q if necessary
return $Q = kP$

Here, we have to perform a post-computation in the accumulators instead of precomputing the points P_i as in the previous approach. Also, the τ endomorphism is applied to the point P, which is usually public. For that reason, we can compute τ with table look-ups instead of performing squarings multiple times.

The downside of this algorithm is that the accumulators carry sensitive information about the digits of the scalar. Also, the accumulators are read and written. As a result, it is necessary to apply the linear pass algorithm to the accumulators Q_i twice per iteration.

5.3 Linear Pass

The linear pass is a method designed to protect sensitive information against side-channel attacks associated with the CPU cache access patterns. Let us consider an array A of size l. Before reading a value $A[i]$, with $i \in [0, l-1]$, the linear pass technique reads the entire array A but only stores, usually into an output register, the requested data $A[i]$. In that way, the attacker does not know which array index was accessed just by analyzing the location of the cache-miss in his own artificially injected data. However, this method causes a considerable overhead, which depends on the size of the array.

In this work, we implemented the linear pass method using 128-bit SSE vectorized instructions and registers. For each array index j, we first copy j to a register and compare this value with the current scalar k τNAF digit. The SSE instruction `pcmpeqq` compares the values of two 128-bit registers A and B and sets the resulting register C with bits one, if A and B are equal, and bits zero otherwise. For that reason, we can use the register C as a mask: if j is equal to the scalar k digit, the register C will contain only bits one. Then, by performing logical operations between C and each of the array values $A[j]$, we can retrieve the requested data.

Experimental results shown that the implementation of the linear pass technique with SSE registers is more efficient than using 64-bit conditional move instructions [25] by a factor of 2.125. The approach just described is presented in Algorithm 9.

Algorithm 9. Linear pass using 128-bit AVX vectorized instructions

Input: An array A of size l, a requested index d, SSE 128-bit registers tmp, dst.
Output: The register dst containing $A[d]$.

1: $dst \leftarrow 0$
2: **for** $i \in \{0, \ldots, l-1\}$ **do**
3: $\quad tmp \leftarrow i$
4: $\quad tmp \leftarrow$ `compare(` tmp, d `)`
($\texttt{compare}$ returns 1^{128} if the operands are equal and 0^{128} otherwise.)
5: $\quad tmp \leftarrow tmp \wedge A[i]$
6: $\quad dst \leftarrow dst \oplus tmp$
7: **end for**

6 Results and Discussion

Our software library can be executed in any Intel platform, which comes with the SSE 4.1 vector instructions and the 64-bit carry-less multiplier instruction

`pclmulqdq`. The benchmarking was executed in an Intel Core i7 4770k 3.50 GHz machine (Haswell architecture) with the TurboBoost and HyperThreading features disabled. Also, the library was coded in the GNU11 C and Assembly languages.

Regarding the compilers, we performed an experimental analysis on the performance of our code compiled with different systems: GCC (Gnu Compiler Collection) versions 5.3, 6.1; and the clang frontend for the LLVM compiler infrastructure versions 3.5 and 3.8. All compilations were done with the flags `-O3 -march=haswell -fomit-frame-pointer`. For the sake of comparison, we reported our timings for all of the previously mentioned compilers. However, when comparing our code with the state-of-the-art works, we opted for the *clang/llvm* 3.8, since it gave us the best performance.

6.1 Parameters

Given $q = 2^m$, with $m = 149$, we constructed our base binary field $\mathbb{F}_q \cong \mathbb{F}_2[x]/(f(x))$ with the 69-term irreducible polynomial $f(x)$ described in Sect. 4. The quadratic extension $\mathbb{F}_{q^2} \cong \mathbb{F}_q[u]/(h(u))$ was built through the irreducible quadratic $h(u) = u^2 + u + 1$. However, our base field arithmetic was computed modulo the redundant trinomial $g(x) = x^{192} + x^{19} + 1$, which has among its irreducible factors, the polynomial $f(x)$.

Our Koblitz curve was defined over $\mathbb{F}_{q^2}$ as $E_1/\mathbb{F}_{q^2} : y^2 + xy = x^3 + ux^2 + u$, and the group $E_1(\mathbb{F}_{q^2})$ contains a subgroup of interest of order

$$r = \texttt{0x637845F7F8BFAB325B85412FB54061F148B7F6E79AE11CC843ADE1470F7E4E29},$$

which corresponds to approximately 255 bits. In addition, throughout our scalar multiplication, we represented the points in λ-affine [20,28] and λ-projective [25] coordinates. We selected an order-r base point P at random represented in λ-affine coordinates.

6.2 Field and Elliptic Curve Arithmetic Timings

In Table 2, we present the timings for the base and the quadratic field arithmetic. The multisquaring operation is used to support the Itoh-Tsujii addition chain, therefore, it is implemented only in the field $\mathbb{F}_{2^{149}}$ (actually, in a 192-bit polynomial in $\mathbb{F}_2[x]$). In addition, we gave timings to reduce a 192-bit polynomial element in $\mathbb{F}_2[x]$ modulo $f(x)$. Finally, all timings of operations in the quadratic field include the subsequent modular reduction.

Applying the techniques presented in [27], we saw that our machine has a margin of error of four cycles. This range is not of significance when considering the timings of the point arithmetic or the scalar multiplication. Nevertheless, for inexpensive functions such as multiplication and squaring, it is recommended to consider it when comparing the timings between different compilers.

Table 2. Timings (in clock cycles) for the finite field operations in $\mathbb{F}_{2^{2 \cdot 149}}$ using different compiler families

Compilers	Multiplication	Squaring	Multi-squaring	Inversion	Reduction modulo $f(x)$
GCC 5.3	52	20	100	2,392	452
GCC 6.1	52	20	104	2,216	452
clang 3.5	64	24	100	1,920	452
clang 3.8	60	20	96	1,894	452

Table 3. The ratio between the arithmetic and multiplication in $\mathbb{F}_{2^{149}}$. The timings were taken from the code compiled with the *clang 3.8* compiler

Operations	Squaring	Multisquaring	Inversion	Reduction modulo $f(x)$
operation / multiplication	0.33	1.60	31.56	7.53

In the following, we compare in Table 3 the base arithmetic operation timings with the multiplication operation, which is the main operation of our library. The ratio squaring/multiplication is relatively expensive. This is because the polynomial $g(x) = x^{192} + x^{19} + 1$, does not admit a reduction specially designed for the squaring operation. Furthermore, the multisquaring and the inversion operations are also relatively costly. A possible explanation is that here, we are measuring timings in a Haswell architecture, which has a computationally cheaper carry-less multiplication when compared with the Sandy Bridge platform [18].

In Table 4 we give the timings of the point arithmetic functions. There, we presented the costs of applying the τ endomorphism to an affine point (two coordinates) and a λ-projective point (three coordinates). The reason is that, depending on the scalar multiplication algorithm, one can apply the Frobenius map on the accumulator (projective) or the base point (affine). In addition, we report in Table 4, the *mixed point doubling* operation, which is defined as follows. Given a point $P = (x_P, y_P)$, the mixed-doubling function computes, $R = (X_R, L_R, Z_R) = 2P$. In other words, it performs a point doubling on an affine point and returns the resulting point in projective representation. Such primitive is useful in the computation of the τNAF representations $\alpha_v = v \bmod \tau^w$ (see Sect. 2.2).

Table 4 also shows the superior performance of the *clang* compiler in the point arithmetic timings, since the only operations where it has a clear disadvantage are the full and mixed point doubling. However, those functions are rarely used throughout a Koblitz curve scalar multiplication. In fact, they are used only in the precomputing phase. Next, in Table 5, we show the relation of the point arithmetic timings with the field multiplication.

Table 4. Timings (in clock cycles) for point addition over a Koblitz curve E_1/q^2 using different compiler families

Compilers	Full Addition	Mixed Addition	Full Doubling	Mixed Doubling	τ endomorphism	
					2 coord	3 coord
GCC 5.3	792	592	372	148	80	120
GCC 6.1	796	588	368	148	80	120
clang 3.5	768	580	404	164	84	124
clang 3.8	752	564	384	160	84	120

Table 5. The ratio between the timings of point addition and the field multiplication. The timings were taken from the code compiled with the *clang 3.8* compiler

Operations	Full Addition	Mixed Addition	Full Doubling	Mixed Doubling	τ endomorphism	
					2 coord	3 coord
operation / multiplication	12.53	9.39	6.40	2.66	1.40	2.00

6.3 Scalar Multiplication Timings

Here the timings for the left-to-right regular w-τNAF τ-and-add scalar multiplication, with $w = 2, 3, 4$ are reported. The setting $w = 2$ is presented in order to analyze how the balance between the pre-computation and the main iteration costs works in practice. Our main result lies in the setting $w = 3$. Also, among the scalar multiplication timings, we show, in Table 6, the costs of the regular recoding and the linear pass functions.

Table 6. A comparison of the scalar multiplication and its support functions timings (in clock cycles) between different compiler families

Compilers	Regular recoding			Linear pass			Scalar multiplication		
	w=2	w=3	w=4	w=2	w=3	w=4	w=2	w=3	w=4
GCC 5.3	1,656	2,740	2,516	8	40	240	100,480	72,556	90,020
GCC 6.1	1,792	2,688	2,480	8	44	240	99,456	71,728	89,740
clang 3.5	1,804	2,680	2,396	8	44	272	96,812	69,696	86,632
clang 3.8	1,808	2,704	2,376	8	40	264	95,196	68,980	85,244

Regarding the regular recoding function, we saw an increase of about 46 % in the 3-τNAF timings when comparing with the $w = 2$ case. The reason is that, for the $w = 3$ case, we must compute a more complicated arithmetic. Also, when selecting the digits, we must perform a linear pass in the array that stores

them. Otherwise, an attacker could learn about the scalar k by performing a timing-attack based on the CPU cache.

The linear pass function also becomes more expensive in the $w = 3$ case, since we have more points in the array. However, in the $m = 149$ case, we have to process 64 more iterations with the width $w = 2$, when compared with the 3-τNAF point multiplication (since the number of iterations depends on m and w: $\frac{m+2}{w-1} + 2$). As a result, the linear pass function overhead is mitigated by the savings in mixed additions and applications of τ endomorphisms in the main loop. Finally, our scalar multiplication measurements consider that the point $Q = kP$ is returned in the λ-projective coordinate representation. If the affine representation is required, it is necessary to add about 2,000 cycles to the total scalar multiplication timings.

6.4 Comparisons

In Table 7, we compare our implementation with the state-of-the-art works. Our 3-τNAF left-to-right τ-and-add point multiplication outperformed by 29.64% the work in [24], which is considered the fastest protected 128-bit secure Koblitz implementation. When compared with prime curves, our work is surpassed by 15.29 % and 13.06 % by the works in [8] and [2], respectively.

Table 7. Scalar multiplication timings (in clock cycles) on 128-bit secure ellitpic curves

Curve/Method	Architecture	Timings
Koblitz over $\mathbb{F}_{2^{283}}$ (τ-and-add, 5-τNAF [24])	Haswell	99,000
GLS over $\mathbb{F}_{2^{2\cdot 127}}$ (double-and-add, 4-NAF [25])	Haswell	61,712
Twisted Edwards over $\mathbb{F}_{(2^{127}-1)^2}$ (double-and-add [8])	Haswell	59,000
Kummer genus-2 over $\mathbb{F}_{2^{127}-1}$ (Kummer ladder [2])	Haswell	60,556
Koblitz over $\mathbb{F}_{4^{149}}$ (τ-and-add, 2-τNAF (this work))	Haswell	96,822
Koblitz over $\mathbb{F}_{4^{149}}$ (τ-and-add, 3-τNAF (this work))	Haswell	69,656
Koblitz over $\mathbb{F}_{4^{149}}$ (τ-and-add, 4-τNAF (this work))	Haswell	85,244

Skylake architecture. In addition, we present timings for our scalar multiplication algorithms, also compiled with *clang* 3.8, in the Skylake architecture (Intel Core i7 6700K 4.00 GHz). The results (in clock cycles) for the cases $w = 2, 3, 4$ are, respectively, 71,138, 51,788 and 66,286.

7 Conclusion

We have presented a comprehensive study of how to implement efficiently Koblitz elliptic curves defined over quaternary fields $\mathbb{F}_{4^m}$, using vectorized instructions on the Intel micro-architectures codename Haswell and Skylake.

As a future work, we plan to investigate the use of 256-bit AVX2 registers to improve the performance of our code. In addition, we intend to implement the scalar multiplication algorithms in other architectures such as the ARMv8. Finally, we would like to design a version of our point multiplication in the multi-core and known point scenarios.

References

1. 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, Heidelberg (2012)
2. Bernstein, D.J., Chuengsatiansup, C., Lange, T., Schwabe, P.: Kummer strikes back: new DH speed records. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8873, pp. 317–337. Springer, Heidelberg (2014)
3. Birkner, P., Longa, P., Sica, F.: Four-dimensional Gallant-Lambert-Vanstone scalar multiplication. Cryptology ePrint Archive, Report 2011/608 (2011). http://eprint.iacr.org/
4. Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., Moeller, B.: Elliptic Curve Cryptography (ECC) cipher suites for Transport Layer Security (TLS). RFC 4492. Internet Engineering Task Force (IETF) (2006). https://tools.ietf.org/html/rfc4492
5. Bluhm, M., Gueron, S.: Fast software implementation of binary elliptic curve cryptography. J. Cryptogr. Eng. **5**(3), 215–226 (2015)
6. Brent, R.P., Zimmermann, P.: Algorithms for finding almost irreducible and almost primitive trinomials. In: Primes and Misdemeanours: Lectures in Honour of the Sixtieth Birthday of Hugh Cowie Williams, Fields Institute, p. 212 (2003)
7. Clift, N.M.: Calculating optimal addition chains. Computing **91**(3), 265–284 (2011)
8. Costello, C., Longa, P.: FourQ: four-dimensional decompositions on a Q-curve over the mersenne prime. In: Iwata, T., et al. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 214–235. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48797-6_10
9. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol version 1.2. RFC 5246. Internet Engineering Task Force (IETF) (2008). https://tools.ietf.org/html/rfc5246
10. Doche, C.: Redundant trinomials for finite fields of characteristic 2. In: Boyd, C., González Nieto, J.M. (eds.) ACISP 2005. LNCS, vol. 3574, pp. 122–133. Springer, Heidelberg (2005)
11. Galbraith, S.D., Gaudry, P.: Recent progress on the elliptic curve discrete logarithm problem. Des. Codes Cryptogr. **78**(1), 51–72 (2016)
12. 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, Heidelberg (2009)
13. Gaudry, P., Hess, F., Smart, N.P.: Constructive and destructive facets of Weil descent on elliptic curves. J. Cryptol. **15**, 19–46 (2002)
14. 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)
15. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer, Secaucus (2003)

16. Hess, F.: Generalising the GHS attack on the elliptic curve discrete logarithm problem. LMS J. Comput. Math. **7**, 167–192 (2004)
17. Intel Corporation: Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Accessed 18 Feb 2016
18. Intel Corporation: Intel 64 and IA-32 Architectures Software Developers Manual 325462–056US (2015)
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. Knudsen, E.W.: Elliptic scalar multiplication using point halving. In: Lam, K.-Y., Okamoto, E., Xing, C. (eds.) ASIACRYPT 1999. LNCS, vol. 1716, pp. 135–149. Springer, Heidelberg (1999)
21. Koblitz, N.: CM-curves with good cryptographic properties. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 279–287. Springer, Heidelberg (1992)
22. Menezes, A., Qu, M.: Analysis of the Weil descent attack of Gaudry, Hess and smart. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 308–318. Springer, Heidelberg (2001)
23. National Institute of Standards and Technology: Recommended elliptic curves for federal government use. NIST Special Publication (1999). http://csrc.nist.gov/csrc/fedstandards.html
24. Oliveira, T., Aranha, D.F., López, J., Rodríguez-Henríquez, F.: Fast point multiplication algorithms for binary elliptic curves with and without precomputation. In: Joux, A., Youssef, A. (eds.) SAC 2014. LNCS, vol. 8781, pp. 324–344. Springer, Heidelberg (2014)
25. Oliveira, T., López, J., Aranha, D.F., Rodríguez-Henríquez, F.: Two is the fastest prime: lambda coordinates for binary elliptic curves. J. Cryptogr. Eng. **4**(1), 3–17 (2014)
26. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169 (2002). http://eprint.iacr.org/
27. Paoloni, G.: How to benchmark code execution times on intel IA-32 and IA-64 instruction set architectures. Technical report, Intel Corporation (2010)
28. Schroeppel, R.: Cryptographic elliptic curve apparatus and method (2000). US patent 2002/6490352 B1
29. Scott, M.: Optimal irreducible polynomials for $GF(2^m)$ arithmetic. Cryptology ePrint Archive, Report 2007/192 (2007). http://eprint.iacr.org/
30. Solinas, J.A.: An improved algorithm for arithmetic on a family of elliptic curves. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 357–371. Springer, Heidelberg (1997)
31. Solinas, J.A.: Efficient arithmetic on Koblitz curves. Des. Codes Cryptogr. **19**(2–3), 195–249 (2000)
32. 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, Heidelberg (2011)
33. Trost, W.R., Xu, G.: On the optimal pre-computation of window τ-NAF for Koblitz curves. Cryptology ePrint Archive, Report 2014/664 (2014). http://eprint.iacr.org/
34. Tsunoo, Y., Tsujihara, E., Minematsu, K., Miyauchi, H.: Cryptanalysis of block ciphers implemented on computers with cache. In: International Symposium on Information Theory and Its Applications, pp. 803–806. IEEE Information Theory Society (2002)

35. Weimerskirch, A., Paar, C.: Generalizations of the Karatsuba algorithm for efficient implementations. Cryptology ePrint Archive, Report 2006/224 (2006). http://eprint.iacr.org/
36. Wenger, E., Wolfger, P.: Solving the discrete logarithm of a 113-bit Koblitz curve with an FPGA cluster. In: Joux, A., Youssef, A. (eds.) SAC 2014. LNCS, vol. 8781, pp. 363–379. Springer, Heidelberg (2014)