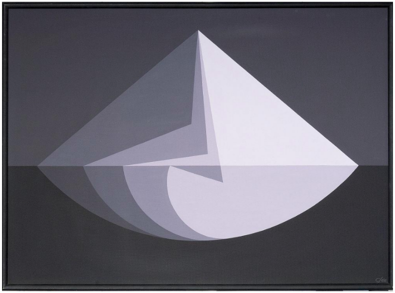
Software implementation of Koblitz curves over quadratic fields

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

[1] Computer Science Department, Cinvestav-IPN
[2] Institute of Computing, University of Campinas



CHES - Santa Barbara, USA
August 18, 2016

Motivation

In this work, we combined the **Koblitz curves**, which allow an efficient scalar multiplication through applications of the Frobenius map, with the **quadratic binary field arithmetic**, that provides opportunities for exploiting the vector instructions available in the current 64-bit high-end architectures, to design a fast **128-bit secure constant-time variable point multiplication**.

Outline

- Koblitz curves over $\mathbb{F}_2$ (brief introduction)
- Koblitz curves over $\mathbb{F}_4$
- Implementation
    - Base field arithmetic
    - Quadratic field arithmetic
    - Scalar multiplication
    - Summary and results

Koblitz curves over $\mathbb{F}_2$

Koblitz curves over $\mathbb{F}_2$

The anomalous binary curves, generally referred to as **Koblitz curves**, are binary elliptic curves proposed for cryptographic use by Neal Koblitz in 1991.

Koblitz curves over $\mathbb{F}_2$

The anomalous binary curves, generally referred to as **Koblitz curves**, are binary elliptic curves proposed for cryptographic use by Neal Koblitz in 1991.

The Weierstrass form of a Koblitz curve is given by

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

Koblitz curves over $\mathbb{F}_2$

The anomalous binary curves, generally referred to as **Koblitz curves**, are binary elliptic curves proposed for cryptographic use by Neal Koblitz in 1991.

The Weierstrass form of a Koblitz curve is given by

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

Since their introduction, the Koblitz curves have been extensively studied for their additional structure that allows a performance speedup in the computation of the scalar multiplication by **replacing point doublings $2(P)$ with the cheaper operation $\tau(P)$** where τ is the Frobenius map $\tau : E_a \to E_a$, defined by

$$\tau(\mathcal{O}) = \mathcal{O}, \qquad \tau(x, y) = (x^2, y^2).$$

Koblitz curves over $\mathbb{F}_2$: τ-adic non-adjacent form

Given a Koblitz curve

$$E_a : y^2 + xy = x^3 + ax^2 + 1,$$

we define $\mu = (-1)^{1-a}$.

Koblitz curves over $\mathbb{F}_2$: τ-adic non-adjacent form

Given a Koblitz curve

$$E_a : y^2 + xy = x^3 + ax^2 + 1,$$

we define $\mu = (-1)^{1-a}$.

The Frobenius map can be seen as a complex number that satisfies

$$\tau^2 + 2 = \mu\tau.$$

As a result, we can multiply points in $E_a(\mathbb{F}_{2^m})$ by elements in $\mathbb{Z}[\tau]$ as

$$(u_{l-1}\tau^{l-1} + \cdots + u_1\tau + u_0)P = u_{l-1}\tau^{l-1}(P) + \cdots + u_1\tau(P) + u_0P.$$

Koblitz curves over $\mathbb{F}_2$: τ-adic non-adjacent form

Given a Koblitz curve

$$E_a : y^2 + xy = x^3 + ax^2 + 1,$$

we define $\mu = (-1)^{1-a}$.

The Frobenius map can be seen as a complex number that satisfies

$$\tau^2 + 2 = \mu\tau.$$

As a result, we can multiply points in $E_a(\mathbb{F}_{2^m})$ by elements in $\mathbb{Z}[\tau]$ as

$$(u_{l-1}\tau^{l-1} + \cdots + u_1\tau + u_0)P = u_{l-1}\tau^{l-1}(P) + \cdots + u_1\tau(P) + u_0 P.$$

In 2000, Jerome Solinas presented a method to represent a scalar k in the form $k' = \sum_{i=0}^{l-1} u_i\tau^i$ with $l \approx m + a$.

Koblitz curves over $\mathbb{F}_2$: summary

Koblitz curves allow the substitution of point doublings with applications of the fast Frobenius map, which results in an **efficient scalar multiplication algorithm**.

In addition, they provide a **rigid curve generation process**.

Koblitz curves over $\mathbb{F}_2$: summary

Koblitz curves allow the substitution of point doublings with applications of the fast Frobenius map, which results in an **efficient scalar multiplication algorithm**.
In addition, they provide a **rigid curve generation process**.

In 2000, the Koblitz curves $E_1/\mathbb{F}_{2^{163}}$ (K-163), $E_0/\mathbb{F}_{2^{233}}$ (K-233), $E_0/\mathbb{F}_{2^{283}}$ (K-283), $E_0/\mathbb{F}_{2^{409}}$ (K-409) and $E_0/\mathbb{F}_{2^{571}}$ (K-571) were standardized by the National Institute of Standards and Technology (NIST) [FIPS 186-2].

Koblitz curves over $\mathbb{F}_2$: summary

Koblitz curves allow the substitution of point doublings with applications of the fast Frobenius map, which results in an **efficient scalar multiplication algorithm**.
In addition, they provide a **rigid curve generation process**.

In 2000, the Koblitz curves $E_1/\mathbb{F}_{2^{163}}$ (K-163), $E_0/\mathbb{F}_{2^{233}}$ (K-233), $E_0/\mathbb{F}_{2^{283}}$ (K-283), $E_0/\mathbb{F}_{2^{409}}$ (K-409) and $E_0/\mathbb{F}_{2^{571}}$ (K-571) were standardized by the National Institute of Standards and Technology (NIST) [FIPS 186-2].

However, since the group of points $E(\mathbb{F}_{2^m})$ is defined over a prime extension field, its arithmetic is costly in modern desktops. Furthermore, in order to design a 128-bit secure point multiplication, we must choose an extension $\tilde{m} \in \{277, 283\}$. The groups $E(\mathbb{F}_{2^{\tilde{m}}})$ contain prime subgroups of order > 254.

Table: Largest prime $E(\mathbb{F}_{2^m})$ subgroup order (bits)

m	$E_0(\mathbb{F}_{2^m})$	$E_1(\mathbb{F}_{2^m})$
251	113	200
257	222	163
263	149	74
269	205	181
271	116	194
277	**275**	**263**
283	**281**	**282**

Koblitz curves over $\mathbb{F}_4$

Koblitz curves over $\mathbb{F}_4$: introduction

The Weierstrass form of a Koblitz curve defined over $\mathbb{F}_4$ is given by

$$E_a : y^2 + xy = x^3 + a\gamma x^2 + \gamma, \text{ with } a \in \{0, 1\}.$$

Here, $\gamma \in \mathbb{F}_4$ satisfies $\gamma^2 = \gamma + 1$.

Koblitz curves over $\mathbb{F}_4$: introduction

The Weierstrass form of a Koblitz curve defined over $\mathbb{F}_4$ is given by

$$E_a : y^2 + xy = x^3 + a\gamma x^2 + \gamma, \text{ with } a \in \{0, 1\}.$$

Here, $\gamma \in \mathbb{F}_4$ satisfies $\gamma^2 = \gamma + 1$.

The Frobenius map $\tau : E_a \to E_a$ is defined by

$$\tau(\mathcal{O}) = \mathcal{O} \qquad \tau(x, y) = (x^4, y^4).$$

Let us consider $\mu = (-1)^a$. Then the Frobenius map can be seen as a complex number that satisfies $\tau^2 + 4 = \mu\tau$.

Koblitz curves over $\mathbb{F}_4$: introduction

The Weierstrass form of a Koblitz curve defined over $\mathbb{F}_4$ is given by

$$E_a : y^2 + xy = x^3 + a\gamma x^2 + \gamma, \text{ with } a \in \{0, 1\}.$$

Here, $\gamma \in \mathbb{F}_4$ satisfies $\gamma^2 = \gamma + 1$.

The Frobenius map $\tau : E_a \to E_a$ is defined by

$$\tau(\mathcal{O}) = \mathcal{O} \qquad \tau(x, y) = (x^4, y^4).$$

Let us consider $\mu = (-1)^a$. Then the Frobenius map can be seen as a complex number that satisfies $\tau^2 + 4 = \mu\tau$.

A Koblitz curve over $\mathbb{F}_4$ has almost-prime group if $E_a(\mathbb{F}_{4^m}) = hn$, where n is prime and $h = \{4, 6\}$, since $\#E_0(\mathbb{F}_4) = 4$ and $\#E_1(\mathbb{F}_4) = 6$.

Koblitz curves over $\mathbb{F}_4$: $E_a(\mathbb{F}_{4^m})$ group order

In order to implement an efficient 128-bit secure scalar multiplication in a 64-bit architecture, our base field size should be at most 192 bits (three 64-bit words). For that reason, we considered primes $m \in \{127, \ldots, 191\}$.

Table: Largest prime $E(\mathbb{F}_{4^m})$ subgroup order (bits)

m	$E_0(\mathbb{F}_{4^m})$	$E_1(\mathbb{F}_{4^m})$
127	196	209
131	108	205
137	173	181
149	239	**255**
151	131	140
157	186	224
163	**324**	189
167	213	**331**
173	196	**308**
179	**272**	196
181	**348**	131
191	173	**362**

Koblitz curves over $\mathbb{F}_4$: $E_a(\mathbb{F}_{4^m})$ group order

In order to implement an efficient 128-bit secure scalar multiplication in a 64-bit architecture, our base field size should be at most 192 bits (three 64-bit words). For that reason, we considered primes $m \in \{127, \ldots, 191\}$.

Table: Largest prime $E(\mathbb{F}_{4^m})$ subgroup order (bits)

m	$E_0(\mathbb{F}_{4^m})$	$E_1(\mathbb{F}_{4^m})$
127	196	209
131	108	205
137	173	181
149	239	**255**
151	131	140
157	186	224
163	**324**	189
167	213	**331**
173	196	**308**
179	**272**	196
181	**348**	131
191	173	**362**

We selected the group $E_1(\mathbb{F}_{4^{149}})$. The factorization of $\#E_1(\mathbb{F}_{4^{149}})$ is given by

6·1886501744269·**44991476563317830182537451551889394335850807098205993761800530540007335546409**.

Koblitz curves over $\mathbb{F}_4$: τ-adic non-adjacent form

Minor changes are required to adapt Solinas' algorithm for representing the scalar k in $\mathbb{Z}[\tau]$.

Koblitz curves over $\mathbb{F}_4$: τ-adic non-adjacent form

Minor changes are required to adapt Solinas' algorithm for representing the scalar k in $\mathbb{Z}[\tau]$.

Window methods can be implemented by computing a Joye-Tunstall-based regular recoding. For a given width-w, we need to precompute $2^{(2w-3)}$ points.

Koblitz curves over $\mathbb{F}_4$: τ-adic non-adjacent form

Minor changes are required to adapt Solinas' algorithm for representing the scalar k in $\mathbb{Z}[\tau]$.

Window methods can be implemented by computing a Joye-Tunstall-based regular recoding. For a given width-w, we need to precompute $2^{(2w-3)}$ points.

Table: Representations of $\alpha_v = v \bmod \tau^w$, for $w \in \{2, 3\}$ and curve E_1

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

Precomputation cost: $1D + 1FA$ $(w = 2)$, $1D + 4FA + 3MA + 4\tau$ $(w = 3)$, $1D + 20FA + 11MA + 5\tau$ $(w = 4)$.

Koblitz curves over $\mathbb{F}_4$: summary

Koblitz curves over $\mathbb{F}_4$ combine the effectiveness of the Frobenius map with the parallelism opportunities offered by the quadratic fields.

Koblitz curves over $\mathbb{F}_4$: summary

Koblitz curves over $\mathbb{F}_4$ combine the effectiveness of the Frobenius map with the parallelism opportunities offered by the quadratic fields.

Contrary to Koblitz curves over $\mathbb{F}_2$, here we have a prime subgroup order of 255 bits, which is suitable for implementing a 128-bit secure scalar multiplication.

Koblitz curves over $\mathbb{F}_4$: summary

Koblitz curves over $\mathbb{F}_4$ combine the effectiveness of the Frobenius map with the parallelism opportunities offered by the quadratic fields.

Contrary to Koblitz curves over $\mathbb{F}_2$, here we have a prime subgroup order of 255 bits, which is suitable for implementing a 128-bit secure scalar multiplication.

Nevertheless, we must consider more carefully the width w of the window methods, since it could result in a costly pre-/post-computation overhead.

Koblitz curves over $\mathbb{F}_4$: summary

Koblitz curves over $\mathbb{F}_4$ combine the effectiveness of the Frobenius map with the parallelism opportunities offered by the quadratic fields.

Contrary to Koblitz curves over $\mathbb{F}_2$, here we have a prime subgroup order of 255 bits, which is suitable for implementing a 128-bit secure scalar multiplication.

Nevertheless, we must consider more carefully the width w of the window methods, since it could result in a costly pre-/post-computation overhead.

Also, the Frobenius map is more expensive (six $\mathbb{F}_{4^m}$ squarings in projective coordinates).

Besides that, to avoid timing attacks, we must not compute the map via look-up tables in the left-to-right point multiplication method.

Implementation

Implementation: preliminaries

Our code was designed for 64-bit platforms provided with SSE4.1 vector instructions and a 64-bit carry-less multiplier. The benchmarking was performed in an Intel Core i7 4770k 3.50 GHz machine (Haswell architecture) with the TurboBoost and HyperThreading technologies disabled.

Implementation: preliminaries

Our code was designed for 64-bit platforms provided with SSE4.1 vector instructions and a 64-bit carry-less multiplier. The benchmarking was performed in an Intel Core i7 4770k 3.50 GHz machine (Haswell architecture) with the TurboBoost and HyperThreading technologies disabled.

The library was coded with GNU11 C and Assembly. For the sake of comparison, our code was compiled with different systems: `gcc 5.3`, `6.1`, `clang 3.5`, `3.8`.

In addition, the code was compiled with the flags `-O3 -march=core-avx2 -fomit-frame-pointer`.

Implementation: base field arithmetic

In order to implement an efficient field arithmetic, we must select an irreducible polynomial to construct the binary extension field $\mathbb{F}_{2^{149}}$. This polynomial should allow a fast modular reduction.

Implementation: base field arithmetic

In order to implement an efficient field arithmetic, we must select an irreducible polynomial to construct the binary extension field $\mathbb{F}_{2^{149}}$. This polynomial should allow a fast modular reduction.

There are no trinomials of degree 149 irreducible over $\mathbb{F}_2$.

Implementation: base field arithmetic

In order to implement an efficient field arithmetic, we must select an irreducible polynomial to construct the binary extension field $\mathbb{F}_{2^{149}}$. This polynomial should allow a fast modular reduction.

There are no trinomials of degree 149 irreducible over $\mathbb{F}_2$.

We found 9680 irreducible pentanomials. However, those polynomials make the shift-and-add modular reduction too costly.

$$x^m + x^a + x^b + x^c + 1$$

Implementation: base field arithmetic

In order to implement an efficient field arithmetic, we must select an irreducible polynomial to construct the binary extension field $\mathbb{F}_{2^{149}}$. This polynomial should allow a fast modular reduction.

There are no trinomials of degree 149 irreducible over $\mathbb{F}_2$.

We found 9680 irreducible pentanomials. However, those polynomials make the shift-and-add modular reduction too costly.

$$x^m + x^a + x^b + x^c + 1$$

Cost: four `xors` (min), twelve `xors` and sixteen shifts (max) per shift-and-add reduction step, depending on the values of m, a, b, c.

The number of reduction steps (after a field multiplication or squaring) is determined by the value $\left\lceil \frac{2m}{m-a} \right\rceil$.

Implementation: redundant trinomials

As a result, we resorted to the redundant trinomial strategy introduced by Brent, Zimmermann (2003) and Doche (2005).

Implementation: redundant trinomials

As a result, we resorted to the redundant trinomial strategy introduced by Brent, Zimmermann (2003) and Doche (2005).

The basic idea is to find a non-irreducible trinomial $g(x)$ which factorizes into an irreducible polynomial $f(x)$ of the desirable degree m. The field $\mathbb{F}_{2^m}$ is isomorphic to $\mathbb{F}_2[x]/(f(x))$ and we can perform its arithmetic modulo $g(x)$.

Implementation: redundant trinomials

As a result, we resorted to the redundant trinomial strategy introduced by Brent, Zimmermann (2003) and Doche (2005).

The basic idea is to find a non-irreducible trinomial $g(x)$ which factorizes into an irreducible polynomial $f(x)$ of the desirable degree m. The field $\mathbb{F}_{2^m}$ is isomorphic to $\mathbb{F}_2[x]/(f(x))$ and we can perform its arithmetic modulo $g(x)$.

In the case of elliptic curves, we can perform the operations on point coordinates modulo $g(x)$ and, at the end of the scalar multiplication, we reduce the result point ($Q = kP$) coordinates modulo $f(x)$.

Implementation: redundant trinomials

Since our target architecture is provided with a 64-bit carry-less multiplier, we searched for trinomials up to degree 192 (three 64-bit words).

Implementation: redundant trinomials

Since our target architecture is provided with a 64-bit carry-less multiplier, we searched for trinomials up to degree 192 (three 64-bit words).

We chose the trinomial $x^{192} + x^{19} + 1$, which factorizes into a 69-term irreducible polynomial $f(x)$ of degree 149:

$$
\begin{aligned}
f(x) = {} & x^{149} + x^{146} + x^{143} + x^{141} + x^{140} + x^{139} + x^{138} + x^{137} + x^{129} + x^{123} + x^{122} + \\
& x^{121} + x^{119} + x^{117} + x^{114} + x^{113} + x^{111} + x^{108} + x^{107} + x^{106} + x^{105} + x^{99} + \\
& x^{94} + x^{92} + x^{91} + x^{90} + x^{86} + x^{85} + x^{83} + x^{81} + x^{80} + x^{78} + x^{77} + x^{75} + \\
& x^{71} + x^{70} + x^{68} + x^{67} + x^{65} + x^{64} + x^{63} + x^{54} + x^{53} + x^{51} + x^{49} + x^{48} + \\
& x^{43} + x^{42} + x^{41} + x^{40} + x^{39} + x^{38} + x^{37} + x^{35} + x^{28} + x^{26} + x^{23} + x^{18} + \\
& x^{17} + x^{16} + x^{15} + x^{12} + x^{11} + x^{10} + x^{9} + x^{3} + x^{2} + x + 1.
\end{aligned}
$$

Implementation: redundant trinomials

The polynomial $x^{192} + x^{19} + 1$ offers us the following advantages:

- The difference $192 - 19 = 173 > 128$ allow us to perform the shift-and-add reduction in just two steps, since we perform it through 128-bit SSE vector instructions.
- Since $192 \bmod 64 = 0$, the amount of shifts during a shift-and-add step can be reduced.

Implementation: redundant trinomials

The polynomial $x^{192} + x^{19} + 1$ offers us the following advantages:

- The difference $192 - 19 = 173 > 128$ allow us to perform the shift-and-add reduction in just two steps, since we perform it through 128-bit SSE vector instructions.
- Since $192 \bmod 64 = 0$, the amount of shifts during a shift-and-add step can be reduced.

At the end of the scalar multiplication algorithm, we must reduce polynomials of degree 191 modulo $f(x)$.

Implementation: redundant trinomials

The polynomial $x^{192} + x^{19} + 1$ offers us the following advantages:

- The difference $192 - 19 = 173 > 128$ allow us to perform the shift-and-add reduction in just two steps, since we perform it through 128-bit SSE vector instructions.
- Since $192 \bmod 64 = 0$, the amount of shifts during a shift-and-add step can be reduced.

At the end of the scalar multiplication algorithm, we must reduce polynomials of degree 191 modulo $f(x)$.

Because $f(x)$ is a 69-term polynomial, this reduction is more efficiently performed via the mul-and-add reduction method. The total cost of this final reduction is 460cc (about 7.53 multiplications in $\mathbb{F}_{4^{149}}$).

Implementation: quadratic field arithmetic

The quadratic field $\mathbb{F}_{2^{2\cdot 149}} \cong \mathbb{F}_{2^{149}}[u]/(h(u))$ was constructed with the degree two monic trinomial $h(u) = u^2 + u + 1$.

Implementation: quadratic field arithmetic

The quadratic field $\mathbb{F}_{2^{2\cdot 149}} \cong \mathbb{F}_{2^{149}}[u]/(h(u))$ was constructed with the degree two monic trinomial $h(u) = u^2 + u + 1$.

Let us consider an element $a = (a_0 + a_1 u) \in \mathbb{F}_{2^{2\cdot 149}}$.

The terms,

$$a_0 = \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$$

and

$$a_1 = \mathbf{C'} \cdot x^{128} + \mathbf{B'} \cdot x^{64} + \mathbf{A'}$$

are 192-bit polynomials, stored into six 64-bit words (**A-C**, **A'-C'**).

Implementation: quadratic field arithmetic

The quadratic field $\mathbb{F}_{2^{2\cdot 149}} \cong \mathbb{F}_{2^{149}}[u]/(h(u))$ was constructed with the degree two monic trinomial $h(u) = u^2 + u + 1$.

Let us consider an element $a = (a_0 + a_1 u) \in \mathbb{F}_{2^{2\cdot 149}}$.

The terms,

$$a_0 = \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$$

and

$$a_1 = \mathbf{C'} \cdot x^{128} + \mathbf{B'} \cdot x^{64} + \mathbf{A'}$$

are 192-bit polynomials, stored into six 64-bit words (**A-C**, **A'-C'**).

Also, let us have three 128-bit registers $\mathbf{R_i}$, with $i \in \{0, 1, 2\}$, which can store two 64-bit words each.

Implementation: quadratic field arithmetic

Reminder: $a_0 = \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$ and $a_1 = \mathbf{C'} \cdot x^{128} + \mathbf{B'} \cdot x^{64} + \mathbf{A'}$.

Implementation: quadratic field arithmetic

Reminder: $a_0 = \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$ and $a_1 = \mathbf{C'} \cdot x^{128} + \mathbf{B'} \cdot x^{64} + \mathbf{A'}$.

The usual way to store the 384-bit element $a = (a_0 + a_1 u)$ is,

$$R_0 = \mathbf{B}|\mathbf{A}, \quad R_1 = \mathbf{A'}|\mathbf{C}, \quad R_2 = \mathbf{C'}|\mathbf{B'}.$$

Implementation: quadratic field arithmetic

Reminder: $a_0 = \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$ and $a_1 = \mathbf{C'} \cdot x^{128} + \mathbf{B'} \cdot x^{64} + \mathbf{A'}$.

The usual way to store the 384-bit element $a = (a_0 + a_1 u)$ is,

$$R_0 = \mathbf{B}|\mathbf{A}, \quad R_1 = \mathbf{A'}|\mathbf{C}, \quad R_2 = \mathbf{C'}|\mathbf{B'}.$$

However, after a 192-bit polynomial multiplication, we have a 384-bit element

$$c = \mathbf{F} \cdot x^{320} + \mathbf{E} \cdot x^{256} + \mathbf{D} \cdot x^{192} + \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$$

which is stored into three 128-bit registers. Then, one step of the shift-and-add reduction is depicted as,

Implementation: quadratic field arithmetic

Reminder: $a_0 = \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$ and $a_1 = \mathbf{C'} \cdot x^{128} + \mathbf{B'} \cdot x^{64} + \mathbf{A'}$.

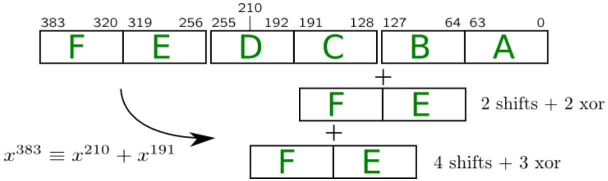
The usual way to store the 384-bit element $a = (a_0 + a_1 u)$ is,

$$R_0 = \mathbf{B}|\mathbf{A}, \quad R_1 = \mathbf{A'}|\mathbf{C}, \quad R_2 = \mathbf{C'}|\mathbf{B'}.$$

However, after a 192-bit polynomial multiplication, we have a 384-bit element

$$c = \mathbf{F} \cdot x^{320} + \mathbf{E} \cdot x^{256} + \mathbf{D} \cdot x^{192} + \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$$

which is stored into three 128-bit registers. Then, one step of the shift-and-add reduction is depicted as,



Cost: (6 shifts + 5 `xor`) × 2 steps × 2 384-bit elem. = 24 shifts + 20 `xor`s.

Implementation: interleaving

Reminder: $a_0 = \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$ and $a_1 = \mathbf{C'} \cdot x^{128} + \mathbf{B'} \cdot x^{64} + \mathbf{A'}$.

Implementation: interleaving

Reminder: $a_0 = \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$ and $a_1 = \mathbf{C'} \cdot x^{128} + \mathbf{B'} \cdot x^{64} + \mathbf{A'}$.

If we consider the interleaving approach, we store the 384-bit element $a = (a_0 + a_1 u)$ as,

$$R_0 = \mathbf{A}|\mathbf{A'}, \quad R_1 = \mathbf{B}|\mathbf{B'}, \quad R_2 = \mathbf{C}|\mathbf{C'}.$$

Implementation: interleaving

Reminder: $a_0 = \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$ and $a_1 = \mathbf{C'} \cdot x^{128} + \mathbf{B'} \cdot x^{64} + \mathbf{A'}$.

If we consider the interleaving approach, we store the 384-bit element $a = (a_0 + a_1 u)$ as,

$$R_0 = \mathbf{A}|\mathbf{A'}, \quad R_1 = \mathbf{B}|\mathbf{B'}, \quad R_2 = \mathbf{C}|\mathbf{C'}.$$

Then, after the quadratic field multiplication, we have two 384-bit elements

$$c = \mathbf{F} \cdot x^{320} + \mathbf{E} \cdot x^{256} + \mathbf{D} \cdot x^{192} + \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$$

and

$$d = \mathbf{F'} \cdot x^{320} + \mathbf{E'} \cdot x^{256} + \mathbf{D'} \cdot x^{192} + \mathbf{C'} \cdot x^{128} + \mathbf{B'} \cdot x^{64} + \mathbf{A'}$$

grouped together, and one step of the shift-and-add reduction is depicted as,

Implementation: interleaving

Reminder: $a_0 = \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$ and $a_1 = \mathbf{C'} \cdot x^{128} + \mathbf{B'} \cdot x^{64} + \mathbf{A'}$.

If we consider the interleaving approach, we store the 384-bit element $a = (a_0 + a_1 u)$ as,

$$R_0 = \mathbf{A}|\mathbf{A'}, \quad R_1 = \mathbf{B}|\mathbf{B'}, \quad R_2 = \mathbf{C}|\mathbf{C'}.$$

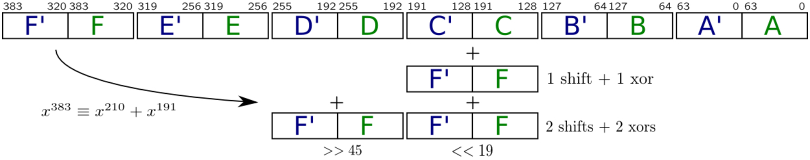
Then, after the quadratic field multiplication, we have two 384-bit elements

$$c = \mathbf{F} \cdot x^{320} + \mathbf{E} \cdot x^{256} + \mathbf{D} \cdot x^{192} + \mathbf{C} \cdot x^{128} + \mathbf{B} \cdot x^{64} + \mathbf{A}$$

and

$$d = \mathbf{F'} \cdot x^{320} + \mathbf{E'} \cdot x^{256} + \mathbf{D'} \cdot x^{192} + \mathbf{C'} \cdot x^{128} + \mathbf{B'} \cdot x^{64} + \mathbf{A'}$$

grouped together, and one step of the shift-and-add reduction is depicted as,



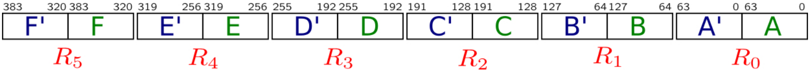
Cost: (3 shifts + 3 xor) $\times$ 3 steps $\times$ 1 384-bit grouped polys = 9 shifts + 9 xors.

Implementation: interleaving

The modular reduction algorithm can be optimized by grouping registers which are shifted by the same value. As a result, we designed a reduction that costs 6 shifts and 9 `xors`.

Implementation: interleaving

The modular reduction algorithm can be optimized by grouping registers which are shifted by the same value. As a result, we designed a reduction that costs 6 shifts and 9 `xors`.

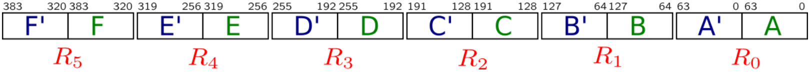


Algorithm Reduction of the terms a_0, a_1 of an element $a \in \mathbb{F}_{2^{2 \cdot 149}}$ modulo $g(x) = x^{192} + x^{19} + 1$

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: $R_7 \leftarrow R_7 \oplus (R_3 \ll 19)$
7: $R_6 \leftarrow R_3 \oplus (R_5 \gg 45)$
8: $R_6 \leftarrow R_6 \oplus (R_6 \ll 19)$
9: $R_6 \leftarrow R_6 \oplus R_0$

Implementation: interleaving

The modular reduction algorithm can be optimized by grouping registers which are shifted by the same value. As a result, we designed a reduction that costs 6 shifts and 9 `xors`.



Algorithm Reduction of the terms a_0, a_1 of an element $a \in \mathbb{F}_{2^{2\cdot 149}}$ modulo $g(x) = x^{192} + x^{19} + 1$

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: $R_7 \leftarrow R_7 \oplus (R_3 \ll 19)$
7: $R_6 \leftarrow R_3 \oplus (R_5 \gg 45)$
8: $R_6 \leftarrow R_6 \oplus (R_6 \ll 19)$
9: $R_6 \leftarrow R_6 \oplus R_0$

In addition, the interleaved representation allows savings in the **precomputing phase of the Karatsuba algorithm**. The drawback of this strategy is the required register reorganization after performing the field multiplication and squaring. However, this penalty is negligible when compared to the savings in the modular reduction algorithm.

Implementation: field arithmetic timings

Table: Field $\mathbb{F}_{2^{2\cdot 149}}$ arithmetic timings (in clock cycles)

Compilers	Multiplication	Squaring	Multisqr. $\mathbb{F}_{2^{149}}$	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: The ratio between the field $\mathbb{F}_{2^{2\cdot 149}}$ arithmetic and multiplication timings

Operations	Squaring	Multisqr. $\mathbb{F}_{2^{149}}$	Inversion	Reduction modulo $f(x)$
operation / multiplication	0.33	1.60	31.56	7.53

Implementation: point arithmetic timings

Table: $E_1(\mathbb{F}_{2^{2\cdot 149}})$ arithmetic timings (in clock cycles)

Compilers	Full Addition	Mixed Addition	Full Doubling	Mixed Doubling	τ endomorphism 2 coord.	τ endomorphism 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: The ratio between the $E_1(\mathbb{F}_{2^{2\cdot 149}})$ arithmetic and the field multiplication timings

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

Implementation: scalar multiplication

Given the Koblitz curve

$$E_1/\mathbb{F}_4 : y^2 + xy = x^3 + ax^2 + a$$

with $a = u$, and its group of points $E_1(\mathbb{F}_{2^{2\cdot 149}})$ which contains a prime subgroup of order ≈ 255 bits, we implemented a **constant-time w-τNAF left-to-right and right-to-left τ-and-add scalar multiplication algorithms.**

Because of the number of points to be pre- (left-and-right approach) or post- (right-to-left approach) computed, we implemented window widths $w \in \{2, 3, 4\}$.

Implementation: scalar multiplication timings

Table: Support functions timings (in clock cycles)

Compilers	Regular recoding			Linear pass		
	w=2	w=3	w=4	w=2	w=3	w=4
GCC 5.3	**1,656**	2,740	2,516	**8**	**40**	**240**
GCC 6.1	1,792	2,688	2,480	**8**	44	**240**
clang 3.5	1,804	**2,680**	2,396	**8**	44	272
clang 3.8	1,808	2,704	**2,376**	**8**	**40**	264

Table: Scalar multiplication timings (in clock cycles)

Compilers	Right-to-Left			Left-to-Right		
	w=2	w=3	w=4	w=2	w=3	w=4
GCC 5.3	98,332	78,248	134,420	100,480	72,556	90,020
GCC 6.1	97,356	79,044	134,152	99,456	71,728	89,740
clang 3.5	**93,260**	**75,812**	140,992	96,812	69,696	86,632
clang 3.8	93,392	77,188	**126,032**	**95,196**	**68,980**	**85,244**

Implementation: summary

The redundant trinomials and interleaving techniques were crucial for achieving our scalar multiplication timings.

Implementation: summary

The redundant trinomials and interleaving techniques were crucial for achieving our scalar multiplication timings.

In addition, the subgroup size allowed us to compute the 128-bit secure point multiplication with an optimal number of τ-and-add iterations.

Implementation: summary

The redundant trinomials and interleaving techniques were crucial for achieving our scalar multiplication timings.

In addition, the subgroup size allowed us to compute the 128-bit secure point multiplication with an optimal number of τ-and-add iterations.

Besides being more expensive in the $\mathbb{F}_4$ case, the Frobenius map is still efficient, costing less than a third of a point doubling operation.

Implementation: summary

The redundant trinomials and interleaving techniques were crucial for achieving our scalar multiplication timings.

In addition, the subgroup size allowed us to compute the 128-bit secure point multiplication with an optimal number of τ-and-add iterations.

Besides being more expensive in the $\mathbb{F}_4$ case, the Frobenius map is still efficient, costing less than a third of a point doubling operation.

The main drawback is the number of points generated by the regular recoding in the $\mathbb{F}_4$ case. The overhead generated by the linear passes and the pre-/post-point computation prevented us from selecting a more agressive value for the window width w.

Table: Pre- and post-computation timings (in clock cycles)

	Right-to-left			**Left-to-right**		
	w=2	**w=3**	**w=4**	**w=2**	**w=3**	**w=4**
pre-/post- comp. cost	3408	13360	49960	3732	9832	32816
% of sc. mult.	3.6	17.3	39.6	3.9	14.2	38.5

Implementation: comparison

Table: 128-bit secure scalar multiplication timings (in clock cycles), Haswell platform

Curve/Method	**Timings**
Koblitz over $\mathbb{F}_{2^{283}}$ (τ-and-add, 5-τNAF [Oliveira *et al.*, 2014])	99,000
GLS over $\mathbb{F}_{2^{2\cdot 127}}$ (double-and-add, 5-NAF [Oliveira *et al.*, 2016])	48,300[1]
Twisted Edwards over $\mathbb{F}_{(2^{127}-1)^2}$ (double-and-add [Costello and Longa, 2015])	56,000
Kummer genus-2 over $\mathbb{F}_{2^{127}-1}$ (Kummer ladder [Bernstein *et al.*, 2014])	60,556
Koblitz over $\mathbb{F}_{4^{149}}$ (left-to-right τ-and-add, 2-τNAF (this work))	96,822
Koblitz over $\mathbb{F}_{4^{149}}$ (left-to-right τ-and-add, 3-τNAF (this work))	**69,656**
Koblitz over $\mathbb{F}_{4^{149}}$ (left-to-right τ-and-add, 4-τNAF (this work))	85,244

[1] New result to be announced in the CHES rump session

Implementation: comparison

Table: 128-bit secure scalar multiplication timings (in clock cycles), Haswell platform

Curve/Method	**Timings**
Koblitz over $\mathbb{F}_{2^{283}}$ (τ-and-add, 5-τNAF [Oliveira *et al.*, 2014])	99,000
GLS over $\mathbb{F}_{2^{2\cdot 127}}$ (double-and-add, 5-NAF [Oliveira *et al.*, 2016])	48,300[1]
Twisted Edwards over $\mathbb{F}_{(2^{127}-1)^2}$ (double-and-add [Costello and Longa, 2015])	56,000
Kummer genus-2 over $\mathbb{F}_{2^{127}-1}$ (Kummer ladder [Bernstein *et al.*, 2014])	60,556
Koblitz over $\mathbb{F}_{4^{149}}$ (left-to-right τ-and-add, 2-τNAF (this work))	96,822
Koblitz over $\mathbb{F}_{4^{149}}$ (left-to-right τ-and-add, 3-τNAF (this work))	**69,656**
Koblitz over $\mathbb{F}_{4^{149}}$ (left-to-right τ-and-add, 4-τNAF (this work))	85,244

[1] New result to be announced in the CHES rump session

Our 3-τNAF left-to-right implementation is the fastest implementation of a 128-bit secure scalar multiplication on a Koblitz curve, surpassing the work of Oliveira *et al.* by 29.6%.

Implementation: comparison

Table: 128-bit secure scalar multiplication timings (in clock cycles), Haswell platform

Curve/Method	Timings
Koblitz over $\mathbb{F}_{2^{283}}$ (τ-and-add, 5-τNAF [Oliveira *et al.*, 2014])	99,000
GLS over $\mathbb{F}_{2^{2\cdot 127}}$ (double-and-add, 5-NAF [Oliveira *et al.*, 2016])	48,300[1]
Twisted Edwards over $\mathbb{F}_{(2^{127}-1)^2}$ (double-and-add [Costello and Longa, 2015])	56,000
Kummer genus-2 over $\mathbb{F}_{2^{127}-1}$ (Kummer ladder [Bernstein *et al.*, 2014])	60,556
Koblitz over $\mathbb{F}_{4^{149}}$ (left-to-right τ-and-add, 2-τNAF (this work))	96,822
Koblitz over $\mathbb{F}_{4^{149}}$ (left-to-right τ-and-add, 3-τNAF (this work))	**69,656**
Koblitz over $\mathbb{F}_{4^{149}}$ (left-to-right τ-and-add, 4-τNAF (this work))	85,244

[1] New result to be announced in the CHES rump session

Our 3-τNAF left-to-right implementation is the fastest implementation of a 128-bit secure scalar multiplication on a Koblitz curve, surpassing the work of Oliveira *et al.* by 29.6%.

In addition, it is competitive to other 128-bit secure point multiplication implementations on binary and prime curves.

Implementation: comparison

Table: 128-bit secure scalar multiplication timings (in clock cycles), Haswell platform

Curve/Method	Timings
Koblitz over $\mathbb{F}_{2^{283}}$ (τ-and-add, 5-τNAF [Oliveira *et al.*, 2014])	99,000
GLS over $\mathbb{F}_{2^{2\cdot 127}}$ (double-and-add, 5-NAF [Oliveira *et al.*, 2016])	48,300[1]
Twisted Edwards over $\mathbb{F}_{(2^{127}-1)^2}$ (double-and-add [Costello and Longa, 2015])	56,000
Kummer genus-2 over $\mathbb{F}_{2^{127}-1}$ (Kummer ladder [Bernstein *et al.*, 2014])	60,556
Koblitz over $\mathbb{F}_{4^{149}}$ (left-to-right τ-and-add, 2-τNAF (this work))	96,822
Koblitz over $\mathbb{F}_{4^{149}}$ (left-to-right τ-and-add, 3-τNAF (this work))	**69,656**
Koblitz over $\mathbb{F}_{4^{149}}$ (left-to-right τ-and-add, 4-τNAF (this work))	85,244

[1] New result to be announced in the CHES rump session

Our 3-τNAF left-to-right implementation is the fastest implementation of a 128-bit secure scalar multiplication on a Koblitz curve, surpassing the work of Oliveira *et al.* by 29.6%.

In addition, it is competitive to other 128-bit secure point multiplication implementations on binary and prime curves.

Skylake timings (left-to-right): 71,138 (w=2), 51,788 (w=3), 66,286 (w=4).

Thank you!

Any questions?