Compact Variable-base ECC Scalar Multiplication using Euclidean Addition Chains

Fabien Herbaut[1], Nicolas Méloni[2] and Pascal Véron[2]
[1]*INSPE Nice-Toulon, Université Côte d'Azur, Institut de Mathématiques de Toulon, France*
[2]*Institut de Mathématiques de Toulon, Université de Toulon, France*

Keywords: Elliptic Curves, Scalar Multiplication, Side-channel Protection, Memory Usage, Addition Chain.

Abstract: The random generation of Euclidean addition chains fits well with a GLV context (Dosso et al., 2018) and provides a method with decent performance despite the growth of the base field required to get the same level of security. The aim of this paper is to reduce the size of the base field required. Combined with an algorithmic improvement, we obtain a reduction of 21% of the memory usage. Hence, our method appears to be one of the most compact scalar multiplication procedure and is particularly suitable for lightweight applications.

1 INTRODUCTION AND CONTEXT

The increasing importance of smart devices such as smart cards or sensor networks comes with growing needs for low cost algorithms in the area of cryptographic primitives. Relevant solutions should involve low memory usage and a compact code which does not sacrifice protection against physical attacks. The aim of this paper is to improve a regular and constant time method, so we obtain a very compact algorithm both from a memory usage and size code point of view. This algorithm could be dedicated for lightweight applications.

1.1 Purpose and Main Idea

The Euclidean addition chains are sequences of couples of integers starting from $(1,1)$ and such that (u,v) leads to $(u,u+v)$ or to $(v,u+v)$. Such chains enable to take advantage of the ZADD operation, which is an efficient way to compute the sum of two points of an elliptic curve with the same Z Jacobian coordinate.

According to (Dosso et al., 2018), this approach fits well with the GLV-like context of an elliptic curve endowed with an efficiently computable endomorphism. Indeed, the method described in (Dosso et al., 2018) leads to a simple and compact way to compute the elliptic curve scalar multiplication kP (ECSM) for a Diffie-Hellman key exchange protocol. The main drawback of this method is that it requires the use of a larger curve as compared to other methods for the same security level. But despite the growth of the base field, it provides decent speed results and very good memory usage performance.

The aim of this paper is to improve the method described in Section 4 of (Dosso et al., 2018). First, we obtain an asymptotic reduction of 7.97% of the size of the base field required. Next, we modify the original ZADD algorithm to save one register over $\mathbb{F}_p$. For practical usage, we obtain a reduction of 21% of the memory used in the method described in (Dosso et al., 2018).

In comparison with the method proposed in (Dosso et al., 2018), the progress lies in the restriction of the random generation to a subset $\mathcal{S}$ of all Euclidean addition chains. Call the chaining $(u,v) \to (u,u+v)$ a small step and the chaining $(u,v) \to (v,u+v)$ a big step. The subset $\mathcal{S}$ to which we restrict ourselves is the one of Euclidean chains with more small steps than big steps. The subset $\mathcal{S}$ has been chosen such that its chains are sufficiently numerous with respect to their growth (see Theorem 1), and quite easy to randomly generate (see lines 2-6 of Algorithm 3).

1.2 Context

Let us now describe more precisely the context of our work. We intend to define a variable scalar multiplication algorithm to be used in the context of Diffie-Hellman key exchange, to provide a regular and constant time algorithm (which is nowadays a minimum requirement for any cryptographic primitive), to re-

Herbaut, F., Méloni, N. and Véron, P.
Compact Variable-base ECC Scalar Multiplication using Euclidean Addition Chains.
DOI: 10.5220/0010551705310539
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 531-539
ISBN: 978-989-758-524-1

spect low memory constraints, to work with an elliptic curve E defined over a prime field $\mathbb{F}_p$ and endowed with one efficient endomorphism.

1.3 Related Papers

One wants to compute a scalar multiplication point kP to perform a part of a Diffie-Hellman key exchange under the above constraints. In (Herbaut et al., 2010) one suggests to randomly sample an Euclidean additions chain $(P,2P) \xrightarrow{small\ step} (P,3P) \xrightarrow{big\ step} (3P,4P) \rightarrow \ldots$ rather than an integer k.

In (Gallant et al., 2001) it is proposed to make use of one efficient endomorphism ϕ on a curve to perform kP. The idea to randomly sample Euclidean additions chains starting from $(P,\phi(P))$ appears in (Dosso et al., 2018). These ideas and the notations are progressively recalled in Section 2.

Note that the idea to randomly sample other elements than the scalar in order to compute scalar multiplication also appears in other works, (see for example the beginning of Section 4 in (Costello et al., 2014)). It necessary leads to a study of the repartition of the computed points (see for example Proposition 4 of (Costello et al., 2014)). The methods described in (Herbaut et al., 2010) leans on an injectivity result (Proposition 3) or an asymptotic result (Theorem 1). As for the main method described in (Dosso et al., 2018), it leans on the injectivity result of Proposition 4. We recall this statement in Section 2, and we improve it with Theorem 1 of Section 3.

As we claim very good memory management, we try to provide a fair comparison with competitive side-channel attack resistant scalar multiplication algorithms in the context of our paper. We give in Section 4 details about the memory consumption of the following well-known algorithms: endomorphism of the x-line (Costello et al., 2014), FourQ (Costello and Longa, 2015), Ted-glv (Gallant et al., 2001) taking benefits from the sign-aligned column decomposition of the scalar (Faz-Hernández et al., 2015) and the twisted Edwards coordinates system, Ted127-glv4 (Faz-Hernández et al., 2015), the Montgomery ladder algorithm exploiting the benefit of the Curve25519 (Bernstein, 2006).

2 BACKGROUND AND NOTATIONS

2.1 Elliptic Curves with One Efficient Endomorphism and Co-Z Arithmetic

The well known GLV method was introduced in 2001 by Gallant, Lambert and Vanstone in (Gallant et al., 2001). It has given rise to many efficient algorithms in a broader range of contexts (see (Galbraith et al., 2009; Longa and Sica, 2014; Faz-Hernández et al., 2015; Costello et al., 2014) for instance). As in (Gallant et al., 2001) we will consider an elliptic curve E defined over a prime field $\mathbb{F}_p$, endowed with an efficiently computable endomorphism $\phi : E \rightarrow E$. A first list of such curves and endomorphisms is given in Section 2 of (Gallant et al., 2001). We will keep the context and the notations of (F. Sica and M. Ciet and J-J. Quisquater, 2003). Namely, we will fix a point $P \in E$ of order N such that $\#E/N \leqslant 4$, we assume that $X^2 + rX + s \in \mathbb{Z}[X]$ is the characteristic polynomial of ϕ and that $\phi(P)=\lambda P$ with $0 < \lambda < N$.

Recall that if two different points P and Q given by their Jacobian coordinates $(X : Y : Z)$ and $(X' : Y' : Z')$ share the same third coordinate $Z = Z'$ (sometimes called the Z-coordinate) then the ZADD operation introduced in (Méloni, 2007) computes the sum $P+Q$ for the elliptic curve law group at a low cost of 5M+2S. Since, this operation and its variants have attracted attention and yielded efficient scalar multiplication schemes (Longa and Miri, 2008; Hutter et al., 2011; Goundar et al., 2010; Goundar et al., 2011; Baldwin et al., 2012). To be more precise, the ZADD operation enables to recover, at the same cost, representatives of both $P+Q$ and P with the same Z-coordinate, or representatives of both $P+Q$ and Q with the same Z-coordinate.

Lastly, with regard to the memory usage, let us emphasize that the x-only trick introduced by Montgomery in (Montgomery, 1987) fits well with the Co-Z arithmetic context. Indeed, consider the formulas given in (Méloni, 2007) to perform the addition of two points with the same Jacobian Z-coordinate: the computation of the X and Y coordinates of the sum does not involve the Z coordinate of the input. When one does not care about the Z coordinate, one can perform the addition saving one register and one multiplication (so the cost drops down to 4M+2S).

2.2 Random Generation of Euclidean Addition Chains in GLV-like Context

The idea of the random generation of chains is introduced in (Herbaut et al., 2010) when the base point P is fixed. The scope is extended to the context of variable-base scalar multiplication on a curve endowed with an endomorphism in (Dosso et al., 2018). The Proposition 4 of the latter paper is the result we want to improve and exploit in this work. Namely, this proposition states that under some assumptions 2^ℓ different chains of length ℓ compute 2^ℓ different points when applying the Algorithm 2 of (Dosso et al., 2018) and starting from $(P,\phi(P))$. Let us recall what it means.

We introduce $\mathcal{M}_\ell = \{0,1\}^\ell$ in order to represent choices of computations in a sequence of ZADD operations. When dealing with elements of $\mathcal{M}_\ell$ in this context, the vocabulary involves the word chains because of the links with addition chains (see (Brauer, 1939) for example). The words Euclidean addition chains (EAC in the sequel of this paper) are also used because of the link with the Euclidean algorithm (see (Montgomery, 1983) for example).

To introduce Algorithm 1, fix $c \in \mathcal{M}_\ell$ and a point $P \in E$. Start from $(P,\phi(P))$. If $c_0 = 1$, compute $(P,P+\phi(P))$ with ZADD. We call this chaining *a small step*. If $c_0 = 0$, compute $(\phi(P),P+\phi(P))$. We call this chaining *a big step*. In both cases the representatives of the computed points share the same Z-coordinate. In the first case, if $c_1 = 1$, compute $(P,2P+\phi(P))$. If $c_1 = 0$ compute $(P+\phi(P),2P+\phi(P))$. In the second case, if $c_1 = 1$, compute $(\phi(P),P+2\phi(P))$. If $c_1 = 0$ compute $(P+\phi(P),P+2\phi(P))$. Repeat this procedure for each bit c_i. At the end, sum the two components with a last call to ZADD. Now assume that $\phi(P)=\lambda P$ with $\lambda \in \mathbb{Z}$. It naturally leads to a point $(k_1+k_2\lambda)P$ for some integers k_1 and k_2. It leads to the Algorithm 1 which corresponds to the method of Section 4 in (Dosso et al., 2018).

Important Remark. This algorithm makes exactly $n+1$ calls to ZADD, where n is the length of c. As it will be explained in the next section, this size will be determined by the required security level. Hence, once the security level of the protocol is settled, the number of calls to ZADD is constant.

The algorithm requires $P \in E$ such that $\phi(P)=\lambda P$ and $c = (c_1,\ldots,c_\ell) \in \mathcal{M}_\ell$. This algorithm computes kP where $k = (1,\lambda)\prod_{i=1}^{\ell} S_{c_i}\binom{1}{1}$ and where S_0 and S_1 denote the matrices defined as follows.

Algorithm 1: EAC_Point_Mul(c: an addition chain of length n).

Require: $P \in E$ and $c \in \mathcal{M}_n$
Ensure: $Q = \chi_{1,\lambda}(c)P$
1: $(U_1,U_2) \leftarrow (P,\phi(P))$
2: **for** $i = 1 \ldots$ length(c) **do**
3: **if** $c_i = 0$ **then**
4: $(U_1,U_2) \leftarrow ZADD(U_2,U_1)$ [it corresponds to (U_2,U_1+U_2)]
5: **else**
6: $(U_1,U_2) \leftarrow ZADD(U_1,U_2)$ [it corresponds to (U_1,U_1+U_2)]
7: **end if**
8: **end for**
9: $(U_1,U_2) \leftarrow ZADD(U_1,U_2)$
10: **return** $Q = U_2$

Definition 1. $S_0 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ *and* $S_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$.

To exploit this linear algebra point of view we also introduce the following definitions.

Definition 2. *Let* $(a,b) \in \mathbb{N}^2$, $s \in \mathbb{N}^*$ *and* $c = (c_1,\ldots,c_s) \in \mathcal{M}_s$. *We define:*

- $\psi_{a,b}(c) = (a,b)\prod_{i=1}^{s} S_{c_i}$ *and* $\chi_{a,b}(c) = (a,b)\prod_{i=1}^{s} S_{c_i}\binom{1}{1}$
- $\mu(c) = \prod_{i=1}^{s} S_{c_i}\binom{1}{1}$

Now we understand the statement of the Proposition 4 in (Dosso et al., 2018). We recall that N denotes the order of the point P and that X^2+rX+s is the characteristic polynomial of the endomorphism ϕ as mentionned in subsection 2.1.

Proposition 1. ***(Proposition 4 in (Dosso et al., 2018))*** *If* $N > F_{n+2}^2(1+|r|+s)$, *then the* 2^n *chains* $c \in \mathcal{M}_n$ *compute* 2^n *different points when applying Algorithm 1.*

This statement has justified the following procedure: randomly choose $c \in \mathcal{M}_n$ and compute $\chi_{1,\lambda}(c)P$ with the algorithm above. Unfortunately, the condition $N > F_{n+2}^2(1+|r|+s)$ involves the use of a curve of greater size than usual. For instance, for a 128-bit security level, the conditions of Proposition 4 of (Dosso et al., 2018) leads to work over a field of size 358.

3 OUR CONTRIBUTION

3.1 Notations and Preliminary Results

The main idea of this paper is to consider a subset $\mathcal{S} \subset \mathcal{M}_\ell$ and to randomly choose c in $\mathcal{S}$ rather than in $\mathcal{M}_\ell$, so it enables to decrease the minimum size of the curve E. Here the point is to choose a subset

$\mathcal{S}$ large enough and whose elements c yield couples $(k_1,k_2) := \psi_{1,\lambda}(c)$ such that k_1 and k_2 are not too big as compared to the order of P. The subset $\mathcal{S}$ we propose in this work is the subset of the chains $c \in \mathcal{M}_\ell$ whose Hamming weight is greater or equal than half the size ℓ. This subset offers a good tradeoff between the two conditions stated above. Its elements are also easy to produce in a simple and quite symmetric way (see lines 2-6 of Algorithm 3). It leads to the following notations.

Definition 3. *Let ℓ and w be two non-zero integers. We define*

- *$\mathcal{M}_{\ell,w}$ as the set of the elements of $\{0,1\}^\ell$ of Hamming weight w,*
- *$\mathcal{M}_{\ell,\geqslant w}$ as the set of the elements of $\{0,1\}^\ell$ of Hamming weight greater or equal to w.*

Although the following lemma is an easy consequence of a coefficient by coefficient comparison of the couples computed by chains, it will make the reading of the proof of Proposition 2 easier.

Lemma 1. *Let ℓ, m, a and b be four integers such that $0 \leqslant m \leqslant \ell$ and $1 \leq a \leq b$. Then*

$$\max\{\chi_{a,b}(c) \mid c \in \mathcal{M}_{\ell,\geqslant m}\} = \max\{\chi_{a,b}(c) \mid c \in \mathcal{M}_{\ell,m}\}.$$

Proof. Fix a chain c of length ℓ and weight m, and consider any chain $\hat{c}$ obtained from c by changing a 0 into a 1. An induction on the length of the chain enables to prove that the couples arising in the computation of $\chi_{a,b}(c)$ are greater or equal than the the couples arising in the computation of $\chi_{a,b}(\hat{c})$ when comparing coefficients by coefficients.

The next result could be considered as the main ingredient of the work (Dosso et al., 2018) and of this contribution. Indeed, when trying to prove that different chains give rise to different points, it enables to bound k_1 and k_2 rather than bounding $k_1 + k_2\lambda$. This result appears in the section 2.1 of (F. Sica and M. Ciet and J-J. Quisquater, 2003) and it is also stated in Lemma 6 of (Longa and Sica, 2014). Proofs release on the irreductibility of $X^2 + rX + s$ on $\mathbb{Z}[X]$.

Lemma 2. *Let $(k_1,k_2) \in \mathbb{Z} \setminus \{(0,0)\}$. If $k_1 + k_2\lambda \equiv 0 \pmod N$ then*

$$\max(|k_1|,|k_2|) \geq \sqrt{\frac{N}{1+|r|+s}}.$$

3.2 New Theoretical Results

We first want to have an upper bound on the coefficient of $\mu(c)$ when c is a chain of $\mathcal{M}_{2n,\geqslant n}$.

Proposition 2. *Let $n \geqslant 3$ be an integer. Then*

$$\max\{\|\mu(c)\|_\infty \mid c \in \mathcal{M}_{2n,\geqslant n}\} = 8\alpha_{n-2} + 11\beta_{n-2}$$

where $\alpha_m = \frac{(1+\sqrt{2})^m + (1-\sqrt{2})^m}{2}$ *and* $\beta_m = \frac{(1+\sqrt{2})^m - (1-\sqrt{2})^m}{2\sqrt{2}}$.

Proof. Fix an integer $n \geqslant 2$. We begin by proving that $\max\{\|\mu(c)\|_\infty \mid c \in \mathcal{M}_{2n,\geqslant n}\} = \max\{\chi_{1,2}(c) \mid c \in \mathcal{M}_{2n-2,n-2}\}$. Recall that for any couple of integers (x,y) we have

$$S_0\binom{x}{y} = \binom{y}{x+y} \text{ and } S_1\binom{x}{y} = \binom{x+y}{y},$$

so the biggest component of $\mu(c)$ is the sum of the components of $\mu(\hat{c})$ where $\hat{c}$ is obtained from c by removing its first element. Taking into account that the first element of c can be 0 or 1, we obtain that $\max\{\|\mu(c)\|_\infty \mid c \in \mathcal{M}_{2n,\geqslant n}\}$ is the greatest integer between $\max\{\chi_{1,1}(c) \mid c \in \mathcal{M}_{2n-1,\geqslant n}\}$ and $\max\{\chi_{1,1}(c) \mid c \in \mathcal{M}_{2n-1,\geqslant n-1}\}$. Using Lemma 1 we see that one can consider $\mathcal{M}_{2n-1,n}$ and $\mathcal{M}_{2n-1,n-1}$ rather than $\mathcal{M}_{2n-1,\geqslant n}$ and $\mathcal{M}_{2n-1,\geqslant n-1}$. Now, with the same argument as in the proof of Lemma 1 the maximum we look for is $\max\{\chi_{1,1}(c) \mid c \in \mathcal{M}_{2n-1,n-1}\}$. Remark that whatever the first step is a small or a big one, it maps $(1,1)$ to $(1,2)$, so the maximum taken is the maximum between $\max\{\chi_{1,2}(c) \mid c \in \mathcal{M}_{2n-2,n-2}\}$ and $\max\{\chi_{1,2}(c) \mid c \in \mathcal{M}_{2n-2,n-1}\}$. The maximum we look for is the first one for the same reason as above. We conclude making use of (Herbaut et al., 2010, Theorem 2) which gives the maximum of $\chi_{1,2}$.

The next result is our main statement: it enables better memory usage than Proposition 1.

Theorem 1. *Let $n \geqslant 3$ be an integer. We consider an elliptic curve E endowed with an endomorphism ϕ whose characteristic polynomial is $X^2 + rX + s$. Let $P \in E$ of prime order N, if*

$$N \geqslant (8\alpha_{n-2} + 11\beta_{n-2})^2 (1 + |r| + s)$$

then, starting from $(P, \phi(P))$, the 2^{2n-1} chains $c \in \mathcal{M}_{2n,\geqslant n}$ compute 2^{2n-1} different points when applying the Algorithm 1 .

Proof. We follow the proof of Proposition 4 in (Dosso et al., 2018). Suppose that the chains c and c' in $c \in \mathcal{M}_{2n,\geqslant n}$ compute the same point when applying Algorithm 1. If $\binom{k_1}{k_2} = \mu(c)$ and $\binom{k'_1}{k'_2} = \mu(c')$, then one should have $k_1 + k_2\lambda \equiv k'_1 + k'_2\lambda \pmod N$ and so $(k_1 - k'_1) + (k_2 - k'_2)\lambda \equiv 0 \pmod N$. But the integers k_i and k'_i are positive, so with Proposition 2 we get $|k_i - k'_i| < 8\alpha_{n-2} + 11\beta_{n-2}$, hence by hypothesis $|k_i - k'_i| < \sqrt{\frac{N}{1+|r|+s}}$. It remains to apply Lemma 6 of (Longa and Sica, 2014) to obtain $\binom{k_1}{k_2} = \binom{k'_1}{k'_2}$ and

thus $c = c'$, as μ is injective (see Proposition 1 in (Dosso et al., 2018)).

The necessary condition in this theorem is more favourable than the one of Proposition 1. Indeed, let us estimate the field size required for a t-bit security level. For such a security level we need to have $2n-1 \geqslant 2t$ which implies to take $n \geqslant t+1$. Thus, if we take $n = t+1$, Theorem 1 shows that it is sufficient to have

$$N \geqslant (8\alpha_{t-1} + 11\beta_{t-1})^2 (1 + |r| + s). \qquad (1)$$

In the worst case we consider we have $1 + |r| + s = 4$. According to the definition of α and β, the right hand side of the last inequation (1) can be seen as a polynomial of degree $2t-2$ in $(1-\sqrt{2})$ and $(1+\sqrt{2})$. If for first approximation we only consider the terms in $(1+\sqrt{2})$ then the condition becomes

$$N \geqslant 4(1+\sqrt{2})^{2t-2} \left(16 + 121/8 + 44/\sqrt{2}\right),$$

which amounts to choosing N such that $\log_2(N) \geqslant 2.54t - 1.41$. Asymptotically we need $\log_2(N) \geqslant 2.54t$, so as compared to the protocol described in (Dosso et al., 2018) (where $\log_2(N) \geqslant 2.76t$) we asymptotically obtain a gain of 7.97%. We give in Table 1 the order of the point P necessary for a given security level computed from inequation (1). For instance in the case of a 128-bit security level, rather than considering a point P of order 358 as in (Dosso et al., 2018) we can now consider a point of order 331, which enables a gain of 7.5%.

3.3 The ZADDu Algorithm

In our implementation, our priority is to minimize the number of active registers. For this purpose, we use a slightly modified version of the ZADD procedure. We call this procedure ZADDu (see Algorithm 2).

Algorithm 2: ZADDu(X_0,Y_0,X_1,Y_1,Z,A).

1: $A \leftarrow X_1 - X_0$
2: $Z \leftarrow Z.A$
3: $A \leftarrow A^2$
4: $X_0 \leftarrow X_0.A$
5: $A \leftarrow X_1.A$
6: $Y_1 \leftarrow Y_1 - Y_0$
7: $X_1 \leftarrow Y_1^2$
8: $X_1 \leftarrow X_1 - X_0$
9: $X_1 \leftarrow X_1 - A$
10: $A \leftarrow A - X_0$
11: $Y_0 \leftarrow Y_0.A$
12: $A \leftarrow X_0 - X_1$
13: $Y_1 \leftarrow A.Y_1$
14: $Y_1 \leftarrow Y_1 - Y_0$

Compared to ZADD, it saves one register. Indeed, the coordinates of the two input points can be used to store some intermediate results. Hence the register B used in Algorithm 9 of (Dosso et al., 2018) is not necessary. It is important to note that this procedure only uses its five parameters and one auxiliary register to perform all its computations, no extra variable is needed.

With this procedure it is possible to perform a scalar multiplication using Algorithm 3. There are two main differences between this algorithm and Algorithm 1. First, note that Theorem 1 requires that more than half of the bits of the chain c are 1's. In practice it only means that more than half of the operations are small steps. As it is purely arbitrary to consider that a 1 represent a small step, the `if` statement from line 2 to 6 ensure that the bit b will represent a big step and thus $b \oplus 1$ a small step. Secondly, each step of the for loop must compute either ZADDu(P_0, P_1) or ZADDu(P_1, P_0) depending on the current EAC bit. In order to avoid the `if` statement and eliminate a possible cache timing attack, the swap of variables is achieved using the same trick shown in (Düll et al., 2015) (line 9 to 11).

From a memory perspective, it is important to note that the whole scalar multiplication only requires 6 active registers for storing 6 field elements. Most scalar multiplication algorithms require at least so many registers just to store the base point P (that must be kept during the whole process, for instance when using double and add) and the result value Q. If the Z-coordinate is not required by the protocol, it is possible to perform a (X,Y)-only scalar multiplication, saving 1 register as the computation involving Z becomes superfluous (Méloni, 2007).

3.4 First Implementation and Practical Results

We describe here a C implementation of our scalar multiplication scheme. It is worth noting that it is a proof of concept implementation, meaning we do not aim at any speed record, and that is why all field arithmetic operations are performed using the GNU Multi Precision library[1]. We measured the computation time of our method. For practical reasons we have used EAC of length 256 so that we can generate 2^{255} different chains leading to a potential security of 127.5 bits. This is coherent with other standards such as `curve25519` (whose security is 126 bits) (Bernstein, 2006). Based on Table 1, it is sufficient to choose a base field defined by a prime p of 331 bits in order to achieve this level of security. That is why we used the curve $E_{331}(\mathbb{F}_{p_{331}}) : y^2 = x^3 + 3$ with $p_{331} = 2^{331} - 36301$ for our experiments.

We use the endomorphism ϕ: (x,y) $\mapsto$ $(\beta x, y)$

[1] http://github.com/eac-team/eac_scalarmult

Table 1: Field and chain sizes required for a given security level when ϕ satisfies $\phi^2 + r\phi + s = 0$ and $(r,s) = (0,1)/(1,1)/(-1,2)$.

Security level	80	96	112	128	192	256	384
Chain length	162	194	126	258	386	514	770
Field size	209	250	290	331	494	657	982

where β is an element of order 3 so that ϕ satisfies the equation $\phi^2 + \phi + 1 = 0$. With the vocabulary of Section 3 we have $r = s = 1$ so the results of Table 1 do apply.

Measurements were performed on an Intel(R) Core(TM) i5-6500 CPU 3.20GHz using `gcc` 9.3.0 and `gmp` 6.2.1. The test procedure first runs 1000 times the scalar multiplication function to heat the cache. Then 1000 random data sets are used and for each of them, one takes the minimum as well as the median value of the execution clock cycle numbers over a batch of 1000 runs. The performance is the average of all the minimums and median values. Cycle count is done as recommended in (Paoloni, 2010). The results obtained with curve E_{331} are given in table 2.

Table 2: Performance (CPU cycles) for a 128-bit security level.

	Min	Median
EAC	1,397,527	1,400,548
EAC (X,Y)-only	1,199,439	1,201,950

To give the reader an order of idea, there are several benchmarks available on the SUPERCOP site (Bernstein and Lange, 2008) for a Diffie-Helmann exchange on an elliptic curve[2]. All the results correspond to optimized versions of Diffie-Helmann key exchange on different curves. Our implementation is just a proof of concept and relies on the general GNU multiprecision library, hence it cannot be considered as an optimized implementation. However, let us consider as an example the NIST P-256 curve which is an international standard used in OpenSSL. Three different implementations of this curve are available in the SUPERCOP package. The first one, developped by Y. Nawaz and G. Gong (University of Waterloo, Canada), only runs on Sun's Ultrasparc processors, hence we could not perform any evaluation on our platform. The second one (named wbl) has been developped by Watson Ladd (Cloudfare company) from the source code of the NIST P-256 implementation of the OpenSSL project developped by Adam Langley (Google). It contains optimized arithmetic operations and uses Jacobian coordinates (like our protocol). The last one (named ref), developped by Jan Mojzis [3], is a constant-time version which uses Jacobian coordinates and implements low-level fast arithmetic (unlike our proof of concept). We followed the procedure described in the SUPERCOP package to evaluate the *wbl* and *ref* versions. We obtained the following outputs on our platform :

```
REF version :
20210125 ionplatform amd64 20210306
crypto_dh nistp256/timingleaks cycles -
1139366 1140132 1139246 1138687  ...
1145814 1139340 1140060 1139187

WBL version :
20210125 ionplatform amd64 20210306
crypto_dh nistp256/timingleaks cycles -
689569 689551 689520 689250 ...
689827 689520 688843 689639
```

Each value corresponds to the number of cycles required to compute one scalar multiplication. Hence our non-optimized version is competitive with the *ref* version and it is almost half as fast as the *wbl* version. Once again, the purpose of this section is not to claim any speed record but to show that, despite the extra cost in terms of field size, our algorithm remains practical speed wise compared to existing standards.

Algorithm 3: EACsmult(X_0,Y_0, c).

Require: $\beta \in \mathbb{F}_p$, P in affine coordinates
Ensure: Return $Q = (X_1, Y_1, Z)$ the point computed from $(P, \phi(P))$ and the Euclidean addition chain c

```
w ← weight(c)
if w ⩽ length(c)/2 then
    b ← 0
else
    b ← 1
end if
X1,Y1,Z ← X0.β,Y0,1
for i = 1...length(c) do
    A ← (X0 ⊕ X1) × (ci ⊕ b)
    X0,X1 ← X0 ⊕ A,X1 ⊕ A
    A ← (Y0 ⊕ Y1) × (ci ⊕ b)
    Y0,Y1 ← Y0 ⊕ A,Y1 ⊕ A
    ZADDu(X0,Y0,X1,Y1,Z)
end for
ZADDu(X0,Y0,X1,Y1,Z)
return X1,Y1,Z
```

[2] https://bench.cr.yp.to/results-dh.html
[3] https://github.com/janmojzis

4 MEMORY USAGE

The aim of this section is to compare our approach with other well known side-channel attack resistant scalar multiplication algorithms from a memory usage point of view.

We sum up the comparison details in Table 3 for a 128-bit level of security. The first column recalls the size of the prime p (given in bits). Then, the "Inputs" double-column gathers the number of $\mathbb{F}_p$-registers and the corresponding size (given in bytes) of the input data of the scalar multiplication procedure. The "ECSM" (Elliptic Curve Scalar Multiplication) double-column gathers the number of $\mathbb{F}_p$-registers and the corresponding size (given in bytes) of the ancillary data used in addition and doubling formulas. Last, the "Total" column summarizes the sizes (given in bytes) of the whole data involved in the ECSM procedure. For the reader's convenience we now provide some details to better understand the memory count for the different methods.

X-line (Costello et al., 2014). The ECSM procedure takes as inputs four affine elements $x(P)$, $x(Q)$, $x(P-Q)$ and $x(P+Q)$ where $Q=\psi(P)$. Each element has two $\mathbb{F}_{p^2}$ coordinates $(Z:T)$ and each element of $\mathbb{F}_{p^2}$ is encoded into two $\mathbb{F}_p$-registers. To compute kP, the procedure uses the ADD_AFFINE and DBLADD_AFFINE formulas described in (Bernstein and Lange, 2005). However, considered "as is", these formulas are not optimal for memory usage. We provide in Algorithms 4 and 5 modified versions in order to minimize the number of registers. The letter in bracket indicates the name of the register originally used in (Bernstein and Lange, 2005). We notice that 6 supplementary $\mathbb{F}_{p^2}$ elements (i.e 12 $\mathbb{F}_p$-registers) are needed for these formulas. Moreover the whole ECSM procedure requires 4 $\mathbb{F}_p$-registers $(z_0,\ldots,z_3)$ and 3 affine points T_0,T_1,T_2 (i.e 12-$\mathbb{F}_p$ registers)

Algorithm 4: ADD_AFFINE(X_2,Z_2,X_3,Z_3,X_1).

1: $X_5 \leftarrow X_2+Z_2$ (A)
2: $Z_5 \leftarrow X_3-Z_3$ (D)
3: $DA \leftarrow X_5 \times Z_5$
4: $X_5 \leftarrow X_2-Z_2$ (B)
5: $Z_5 \leftarrow X_3+Z_3$ (C)
6: $CB \leftarrow X_5 \times Z_5$
7: $X_5 \leftarrow (DA+CB)^2$
8: $Z_5 \leftarrow X_1 \times (DA-CB)^2$
9: **return** X_5,Z_5

Ted-glv-Sac. To compute kP we exploit the well-known GLV method (Gallant et al., 2001) on a curve defined over $\mathbb{F}_p$ with one efficiently computable endomorphism, and the Sign-Aligned Column decomposition of k decribed in (Faz-Hernández et al., 2015).

Algorithm 5: DBLADD_AFFINE(X_2,Z_2,X_3,Z_3,X_1).

1: $X_4 \leftarrow X_2+Z_2$ (A)
2: $Z_4 \leftarrow X_3-Z_3$ (D)
3: $Z_4 \leftarrow X_4 \times Z_4$ (DA)
4: $X_5 \leftarrow X_2-Z_2$ (B)
5: $Z_5 \leftarrow X_3+Z_3$ (C)
6: $Z_5 \leftarrow X_5 \times Z_5$ (CB)
7: $X_5 \leftarrow X_4^2$ (AA)
8: $BB \leftarrow X_5^2$
9: $E \leftarrow X_5-BB$
10: $X_4 \leftarrow X_5 \times BB$
11: $X_5 \leftarrow (Z_4+Z_5)^2$
12: $Z_5 \leftarrow X_1 \times (Z_4-Z_5)^2$
13: $Z_4 \leftarrow E \times (BB+a24 \times E)$
14: **return** X_4,Z_4,X_5,Z_5

The points are represented using the Twisted Edwards coordinates system. To obtain better performances, point doubling and addition are computed mixing standard twisted Edwards coordinates with extended twisted Edwards coordinates. Standard twisted Edwards coordinates are encoded on 3 $\mathbb{F}_p$-registers while extended twisted Edwards coordinates need 4 resisters. The algorithm is described in (Dosso et al., 2018, Alg. 7)[4].

The point doubling takes as input extended twisted Edwards coordinates and outputs extended twisted Edwards coordinates (for the point Q). The corresponding formula (non optimized for memory) can be found in the section 3.3 of (Hisil et al., 2008). The C implementation shows that in practice only the 4 registers of the point Q and 2 additional registers are needed to update Q. The point addition takes as input extended twisted Edwards coordinates and standard twisted Edwards coordinates. The output is expressed in extended twisted Edwards coordinates. The formula is given in section 3.2 of (Hisil et al., 2008) when $a=-1$ and $Z_2=1$. In the C implementation which is provided, only the 4 registers of the point Q and 2 additional registers are needed to perform this point addition. Hence 6 registers are needed to compute any of these two operations. Last, the input of the ECSM algorithm consists of two points given in standard twisted Edwards coordinates, that is 6 registers.

Ted127-glv4 (Faz-Hernández et al., 2015). Here the authors take benefit of two efficiently computable endomorphisms ϕ and ψ on a curve defined over $\mathbb{F}_{p^2}$. The points are represented either in standard twisted Edwards coordinates or in extended twisted Edwards coordinates. Each coordinate is encoded on 2 $\mathbb{F}_p$-registers. The input of the algorithm consists of four points: P, $\phi(P)$, $\psi(P)$, $\phi\psi(P)$ (we need 4 coordinates for each of these points, see annex B of (Faz-Hernández et al., 2015)). It gives a total of

[4] A C implementation is available on https://github.com/eacElliptic/Scalar-multiplications/tree/master/C/with_edwards_coord/sac_glv

Table 3: Memory storage for a 128-bit security level.

		Inputs		ECSM		
	$log_2(p)$ (bits)	$\mathbb{F}_p$ registers	Size(bytes)	$\mathbb{F}_p$ registers	Size(bytes)	Total
x-line (Costello et al., 2014)	128	16	256	28	448	704
Ted-glv-sac	256	6	192	6	192	384
Ted127-glv4 (Faz-Hernández et al., 2015)	127	32	508	48	762	1270
FourQ(Costello and Longa, 2015)	127	4	64	98	1556	1620
Montgomery(Bernstein, 2006)	256	1	32	6	192	224
EAC (Dosso et al., 2018)	358	5	224	2	90	314
EAC (this paper)	331	5	207	1	42	249 (-21%)
EAC (X,Y)-only (Dosso et al., 2018)	358	4	179	2	90	269
EAC (X,Y)-only (this paper)	331	4	166	1	42	**207 (-23%)**

32 $\mathbb{F}_p$-registers. Three coordinates are used to encode P, which are later extended to 4 coordinates. In the case of variable base scalar multiplication, as explained in (Faz-Hernández et al., 2015), 4 more points are computed during the computation of kP. Each point is represented using 4 coordinates $(X+Y, Y-X, 2Z, 2T)$, which leads to 32 $\mathbb{F}_p$-registers. The results given in table 3 come from (Faz-Hernández et al., 2015) where the formulas of (Hisil et al., 2008) have been adapted to reduce memory storage. The algorithm computes a doubling followed by an addition. The result of these operations is stored in a point Q which is represented by 3 coordinates plus two additional elements of $\mathbb{F}_{p^2}$, namely T_a and T_b (see (Hamburg, 2012) for the description of the Hamburg's "extensible" strategy which requires 10 registers). Doubling needs 2 additional elements of $\mathbb{F}_{p^2}$, hence 4 registers (note that in Algorithm 11 of (Faz-Hernández et al., 2015) X_1 and Z_1 can be used instead of X_2 and Z_2). A point addition requires 3 $\mathbb{F}_{p^2}$-elements, so 6 registers (X_1, Y_1 and Z_1 can be also used instead of X_3, Y_3 and Z_3 if line 10 is computed before line 8 in Algorithm 11 of (Faz-Hernández et al., 2015)). To conclude, doubling and adding can be done with 6 registers for additional elements and 10 registers for the storage of the point Q.

FourQ (Costello and Longa, 2015). The ECSM is processed on an elliptic curve defined over $\mathbb{F}_{p^2}$ and endowed with two efficiently computable endomorphisms. Computations are done using twisted Edward coordinates. Elements of $\mathbb{F}_{p^2}$ are represented using two $\mathbb{F}_p$-registers. [5].

The variable-base scalar multiplication is implemented in the function `ecc_mul` of the `ecpp2.c` file. The input consists of two points given in affine coordinates that require 8 $\mathbb{F}_p$-registers. The function needs a point R with 5 coordinates (X,Y,Z,T_a,T_b) and 9 points (S, T_0,..., T_7) with 4 coordinates $(X+Y, Y-X, 2Z, 2dT)$, which leads to a total of 82 $\mathbb{F}_p$-registers. Notice that doubling a point (function `eccdouble` from `ecpp2_core.c`) only needs two extra elements of $\mathbb{F}_{p^2}$ (i.e 4 registers). Adding two points (function `eccadd` from `ecpp2_core.c`) requires one extra-point R given by its 4 coordinates $(X+Y, Y-Z, Z, T)$ and two elements of $\mathbb{F}_{p^2}$ which leads to a total of 12 registers. The whole ECSM needs 98 $\mathbb{F}_p$-registers.

Montgomery Ladder (Montgomery, 1987). Montgomery curves combined with the Montgomery ladder algorithm give rise to fast and secure ECSM and are of particular interest for elliptic curve cryptography. Since December 2016, the IETF (Internet Engineering Task Force) uses two Montgomery curves (Curve25519 (Bernstein, 2006) and Curve448 (Hamburg, 2015)) for instantiating Diffie-Hellman protocols named X25519 and X448 in the Internet Key Exchange Protocol Version 2 (IKEv2). We focus on Curve25519 which is defined over $\mathbb{F}_p$. Results given in table 3 come from (Düll et al., 2015) where the authors use formulas for the ladder step that minimize the number of temporary (stack) variables (see Algorithm 2 of (Düll et al., 2015)). The ECSM algorithm takes as input the x-coordinate of P (one $\mathbb{F}_p$-register). The algorithm needs 6 registers (X_1, X_2, Y_1, Y_2, T_1 and T_2) to process.

EAC and EAC (X,Y)-only (this Paper). The algorithm takes as input data the points P and ϕ (P). Five registers are required to store these points. For the (X,Y)-only version, only four $\mathbb{F}_p$-registers are required. We stress that in this case the Z-coordinate is not required in the ZADDu procedure. To compute one step of the main loop of Algorithm 3, only one additional register is used in ZADDu in both cases. Notice that one of the main advantage of our method is that after the first iteration of the main loop the input data are not used again. Hence the corresponding registers can be used to store new computed values. This is not the case for the preceding methods.

[5]The results given in Table 3 come from http://research.microsoft.com/en-us/projects/fourqlib/

To conclude this section, let us supply some arguments about the natural compacity of the code induced by our method. First, when compared to the classical algorithms which use endomorphisms, we do not need to implement any decomposition of the scalar k. Second, when compared to methods which need to implement double and addition formulae, our method uses only one operation ZADDu which is easy to implement in a compact way.

REFERENCES

Baldwin, B., Goundar, R. R., Hamilton, M., and Marnane, W. P. (2012). Co-z ECC scalar multiplications for hardware, software and hardware-software co-design on embedded systems. *J. Cryptographic Engineering*, 2(4):221–240.

Bernstein, D. J. (2006). Curve25519: New Diffie-Hellman speed records. In Yung, M., Dodis, Y., Kiayias, A., and Malkin, T., editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bernstein, D. J. and Lange, T. (2005). Explicit-Formulas Database. https://www.hyperelliptic.org/EFD/.

Bernstein, D. J. and Lange, T. (2008). ebacs: Ecrypt benchmarking of cryptographic systems. https://bench.cr.yp.to, accessed 1 march 2021.

Brauer, A. (1939). On addition chains. *Bulletin of the American Mathematical Society*, 45(10):736–739.

Costello, C., Hisil, H., and Smith, B. (2014). Faster compact Diffie–Hellman: Endomorphisms on the x-line. In Nguyen, P. Q. and Oswald, E., editors, *Advances in Cryptology – EUROCRYPT 2014*, pages 183–200, Berlin, Heidelberg. Springer Berlin Heidelberg.

Costello, C. and Longa, P. (2015). Four$\mathbb{Q}$: four-dimensional decompositions on a $\mathbb{Q}$-curve over the Mersenne prime. In *Advances in Cryptology – ASIACRYPT 2015, Auckland, New Zealand* , pages 214–235. Berlin: Springer.

Dosso, Y., Herbaut, F., Méloni, N., and Véron, P. (2018). Euclidean addition chains scalar multiplication on curves with efficient endomorphism. *Journal of Cryptographic Engineering*, 8(4):351–367.

Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C., Sánchez, A. H., and Schwabe, P. (2015). High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Des. Codes Cryptography*, 77(2–3):493–514.

F. Sica and M. Ciet and J-J. Quisquater (2003). Analysis of the Gallant-Lambert-Vanstone method based on efficient endomorphisms: elliptic and hyperelliptic curves. In *Selected Areas in Cryptography*, volume 2595 of *LNCS*, pages 21–36. Springer.

Faz-Hernández, A., Longa, P., and Sánchez, A. H. (2015). Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves (extended version). *J. Cryptographic Engineering*, 5(1):31–52.

Galbraith, S. D., Lin, X., and Scott, M. (2009). Endomorphisms for faster elliptic curve cryptography on a large class of curves. In *Advances in Cryptology - EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 518–535. Springer Berlin Heidelberg.

Gallant, R. P., Lambert, R. J., and Vanstone, S. A. (2001). Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology — CRYPTO*, volume 2139 of *LNCS*, pages 190–200. Springer.

Goundar, R. R., Joye, M., and Miyaji, A. (2010). Co-Z addition formulæ and binary ladders on elliptic curves - (extended abstract). In *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 65–79.

Goundar, R. R., Joye, M., Miyaji, A., Rivain, M., and Venelli, A. (2011). Scalar multiplication on Weierstraß elliptic curves from co-z arithmetic. *Journal of Cryptographic Engineering*, 1(2):161–176.

Hamburg, M. (2012). Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Report 2012/309. https://eprint.iacr.org/2012/309.

Hamburg, M. (2015). Ed448-goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625. https://eprint.iacr.org/2015/625.

Herbaut, F., Liardet, P.-Y., Méloni, N., Téglia, Y., and Véron, P. (2010). Random euclidean addition chain generation and its application to point multiplication. In *INDOCRYPT 2010*, volume 6498, pages 238–261, Hyderabad, India. Springer.

Hisil, H., Wong, K. K.-H., Carter, G., and Dawson, E. (2008). Twisted Edwards curves revisited. In *Advances in Cryptology – ASIACRYPT 2008, Melbourne*, pages 326–343. Berlin: Springer.

Hutter, M., Joye, M., and Sierra, Y. (2011). Memory-constrained implementations of elliptic curve cryptography in co-Z coordinate representation. In *Progress in Cryptology - AFRICACRYPT 2011*, pages 170–187.

Longa, P. and Miri, A. (2008). *New Composite Operations and Precomputation Scheme for Elliptic Curve Cryptosystems over Prime Fields*, pages 229–247. Springer Berlin Heidelberg, Berlin, Heidelberg.

Longa, P. and Sica, F. (2014). Four-dimensional Gallant–Lambert–Vanstone scalar multiplication. *Journal of Cryptology*, 27(2):248–283.

Méloni, N. (2007). New point addition formulae for ECC applications. In *Arithmetic of Finite Fields*, volume 4547 of *LNCS*, pages 189–201. Springer Berlin / Heidelberg.

Montgomery, P. L. (1983). Evaluating Recurrences of form $x_{m+n} = f(x_m, x_n, x_{m-n})$ via Lucas chains. Available at ftp.cwi.nl:/pub/pmontgom/Lucas.ps.gz.

Montgomery, P. L. (1987). Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–243.

Paoloni, G. (2010). How to benchmark code execution times on intel® ia-32 and ia-64 instruction set architectures. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.