# Euclidean addition chains scalar multiplication on curves with efficient endomorphism

Fangan-Yssouf Dosso, Fabien Herbaut, Nicolas Méloni, Pascal Véron

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

# Euclidean addition chains scalar multiplication on curves with efficient endomorphism

**Yssouf Dosso · Fabien Herbaut · Nicolas Méloni · Pascal Véron**

This is a pre-print of an article published in "Journal of Cryptographic Engineering". The final authenticated version is available online at: https://doi.org/10.1007/s13389-018-0190-0.

the date of receipt and acceptance should be inserted later

**Abstract** Random Euclidean addition chain generation has proven to be an efficient low memory and SPA secure alternative to standard ECC scalar multiplication methods in the context of fixed base point [21]. In this work, we show how to generalize this method to random point scalar multiplication on elliptic curves with an efficiently computable endomorphism. In order to do so we generalize results from [21] on the relation of random Euclidean chains generation and elliptic curve point distribution obtained from those chains. We propose a software implementation of our method on various platforms to illustrate the impact of our approach. For that matter, we provide a comprehensive study of the practical computational cost of the modular multiplication when using Java and C standard libraries developed for the arithmetic over large integers.

**Keywords** Addition chains · Co-Z arithmetic · scalar multiplication · GLV · Android

## 1 Introduction

Let $p$ be a prime number. An elliptic curve in short Weierstrass form over a finite prime field $\mathbb{F}_p$ is defined by $E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p | y^2 = x^3 + ax + b\} \cup \mathcal{O}$, with $a, b \in \mathbb{F}_p$ satisfying $4a^3 + 27b^2 \neq 0$ and $\mathcal{O}$ being called the point at infinity. The set $E(\mathbb{F}_p)$ is an abelian group with an efficiently computable group law. The main operation in elliptic curve cryptography is scalar multiplication, that is the computation of $kP$, where $P$ is a prime order point on a curve and $k$ is an integer. Optimizing this operation is directly linked to the problem of finding a short addition chain computing the integer $k$. The most common way to find such chains relies on the classical *double-and-add* algorithm and its many variants and improvements [5,34,29,30]. Another approach consists in using a rather different family of chains, the Euclidean Addition Chains (EAC). If $C = (c_1, \ldots, c_l)$ is an EAC computing $k$, one can compute $kP$ in $l$ differential point additions, that is additions for which the difference of the two summands is already known. On elliptic curves in short Weierstrass form it provides an efficient, low memory and simple side channel attack (SSCA) resistant method to perform scalar multiplication when combined with Co-Z arithmetic, as long as one is capable of finding a chain of small length [28]. However, given a large integer $k$, it appears to be quite time consuming to find suitable chains. A natural way to bypass that issue is to randomly generate a small EAC and to consider the corresponding integer. However, in that case, one does not fully control the distribution of the corresponding points. To obtain a proven security of $n$ bits, that is to say to be able to guarantee that one can generate $2^n$ different chains computing $2^n$ different points, one can work with chains of length $n$ but has to pre-compute the pair $(F_{n+2}P, F_{n+3}P)$, where $\{F_n\}$ is the Fibonacci sequence, and work on larger fields than standard methods. Those constraints limit the use of this approach to the case of fixed base point scalar multiplication [21].

Y. Dosso, N. Méloni, P. Véron
Institut de Mathématiques de Toulon
Université de Toulon, France
E-mail: dosso@univ-tln.fr
E-mail: meloni@univ-tln.fr
E-mail: veron@univ-tln.fr

F. Herbaut
Université de Nice Sophia Antipolis, France
Institut de Mathématiques de Toulon
Université de Toulon, ESPE Nice-Toulon
E-mail: herbaut@unice.fr

In this paper, we propose to generalize that previous work to the case of random base point scalar multiplication on elliptic curves with an efficiently computable endomorphism. Moreover, we want to derive from this generalization a practical implementation which fits the following constraints.

**SSCA resistance:** we need to design a regular and constant-time algorithm to be protected against simple side channel attacks.

**Cache timing attack resistance:** the execution flow (sequence of instructions) must be independent from the key used in order to avoid recent cache instructions attacks [1]. Data loaded into cache must also be independent from the key in order to avoid data cache attacks [3].

**Low memory:** we want to minimize the number of registers needed to store the coordinates of the various points involved in the computation of $kP$. For resource constrained devices (like IoT devices), it is of utmost importance to design a low memory algorithm with little impact on the performances of the scalar multiplication operation.

**Non specific libraries dependencies:** in order to manage arithmetic over large integers, we only consider general purpose multi-precision libraries with long term support (BigInteger library for Java-based platforms and GNU Multiple Precision library for other platforms).

**Curves with one efficient endomorphism:** in order to design an efficient algorithm, we consider curves with one endomorphism. Taking into account our memory usage constraint, we focus on curves with exactly one endomorphism. Indeed, the use of extra endomorphisms leads to extra precomputation stages, and so to extra memory usage. As an example, a secure implementation of a scalar multiplication algorithm using a curve with one endomorphism needs to precompute two points. For a curve, with two endomorphisms, the same implementation needs eight points [12].

**Curves over $\mathbb{F}_p$:** to be compliant with actual elliptic curve cryptography standards, we focus on the case of an elliptic curve defined over $\mathbb{F}_p$. Moreover, to be more resource friendly, we do not want to have to deal both with arithmetic modulo $p$ and arithmetic modulo $p^2$.

In the sequel, we first recall the necessary background on EAC, Co-Z arithmetic and elliptic curves with fast endomorphism (Section 2). Then we generalize the results from [21] on the distribution of integers computed by an EAC starting from any pair of points $(aP, bP)$ when $P$ is fixed (Corollary 1). Finally we consider the case of scalar multiplication with a random base point on curves with a fast endomorphism $\phi$ (Proposition 4). We show that under some assumptions we can guarantee a given security when starting from a pair of points $(P, \phi(P))$. We derive from those results a new scalar multiplication scheme on curves with a fast endomorphism (Section 4). The complexity analysis of this scheme shows that it can be competitive with state of the art methods depending on the relative costs of modular multiplication on fields of various sizes. To illustrate our point, we propose software implementations on various platforms using standard libraries for arithmetic over large integers. We discuss the efficiency of our method from a speed and memory consumption point of view (Section 5, 6, 7 and 8).

## 2 Background on ECC scalar multiplication

### 2.1 Curve with efficient endomorphism

In 2001, Gallant, Lambert and Vanstone introduced a new approach to speed up scalar multiplication on elliptic curves with an efficiently computable endomorphism [15], the so called GLV method. Let $E$ be an elliptic curve over $\mathbb{F}_p$ such that $\#E(\mathbb{F}_p) = N \times h$, where $N$ is a large prime and $h$ a small co-factor (i.e. 1, 2 or 4). Let $\phi$ be a non trivial endomorphism. Then there exists $\lambda$ such that for all points $P$ of order $N, \phi(P) = \lambda P$. Now let us consider a scalar $k \in [1, N-1]$. It has been proven that one can always find $k_1, k_2$ such that $k \equiv k_1 + k_2\lambda \mod N$ and $\max\{|k_1|, |k_2|\} \leq c\sqrt{N}$ for some computable constant $c$ [11]. On curves with such an endomorphism, $kP$ can be computed by performing a multi-scalar multiplication saving half the point doublings in exchange of a few point additions. The standard method consists of storing $P$, $\phi(P)$, $P + \phi(P)$ and $P - \phi(P)$ (in addition of the current point) but is vulnerable to simple side channel attack. To prevent such attacks, the most recent implementations use a combination of Least Significant Bit set representation and sign alignment [12]. It has the advantages to make the scalar multiplication regular (one doubling and one addition per scalar bit) and to reduce the storage requirement as only the points $P$ and $P + \phi(P)$ need to be stored. The GLV method has later been extended to a larger set of curves defined over $\mathbb{F}_{p^2}$ [14,32,19,27] which are endowed by more than one endomorphism. In this case additional performance gains can be achieved, whereas it implies the need of more memory.

### 2.2 Co-Z arithmetic on elliptic curves

Fast elliptic curve computations has become an important research area over the past years. Many formulas,

coordinate systems or curve shapes have been proposed in order to implement the associated group law. For a comprehensive overview, one can refer to [8,20]. The traditional approach is to consider curves in Jacobian coordinates, a point on such a curve is represented as a triple $(X : Y : Z)$ or any triple $(\alpha^2 X : \alpha^3 Y : \alpha Z)$, with $\alpha \in \mathbb{F}_p^*$. In that case, the formulae given in [28] enable to compute the sum of two points, $P$ and $Q$, sharing the same $Z$-coordinate, lowering the computational cost from 11M+5S for a standard point addition to 5M+2S. At the same cost one obtains the coordinates of a point $\tilde{P}$ such that :

- $P + Q$ and $\tilde{P}$ share the same Z-coordinate,
- $P$ and $\tilde{P}$ are in the same equivalent co-set.

This operation is sometimes called ZADD, or ZADDU ([16,18]) to say ZADD with Update. Several works have used those formulae to propose efficient and secure scalar multiplication schemes: one can see [23,2,21] for right-to-left algorithms and [26] for left-to-right applications.

### 2.3 Euclidean addition chains

Given an ordered pair of points $(P, Q)$ sharing the same $Z$-coordinate, one can compute, using the ZADD operation, either $(Q, P + Q)$ or $(P, P + Q)$ with the same $Z$-coordinate. Following notations from [21], the first computation will be called a *big step* (denoted by 0) and the second one will be called a *small step* (denoted by 1). For instance, starting from $P$ and $2P$ (sharing the same $Z$-coordinate), one can compute $(P, 3P)$ or $(2P, 3P)$. Then, one can obtain $(P, 4P)$ or $(3P, 4P)$ from $(P, 3P)$, and $(2P, 5P)$ or $(3P, 5P)$ from $(2P, 3P)$, and so on. One can thus perform a whole scalar multiplication using Algorithm 1 and the ZADD operation (see Definition 1 below for the definition of $\chi(c)$).

**Definition 1**

. An Euclidean addition chain (or EAC) of length $s$ is a finite sequence $(c_i)_{i=1\ldots s}$ of elements of $\{0, 1\}$.
. We will denote the set of EAC by $\mathcal{M}$ and the set of EAC of length $s$ by $\mathcal{M}_s$.
. To such a sequence we associate $(v_i, u_i)_{i=0..s}$ a sequence of elements of $\mathbb{N}^2$ defined as follows:
  - $(v_0, u_0) = (1, 2)$,
  - $\forall i \in [\![1, s]\!]$, $(v_i, u_i) = (v_{i-1}, v_{i-1} + u_{i-1})$ if $c_i = 1$ (small step),
  - $\forall i \in [\![1, s]\!]$, $(v_i, u_i) = (u_{i-1}, v_{i-1} + u_{i-1})$ if $c_i = 0$ (big step).
. We will say that the sequence or the EAC $(c_i)_{i=1\ldots s}$ computes the integer $v_s + u_s$ and the pair $(v_s, u_s)$. If $c = (c_i)_{i=1\ldots s}$ we will denote the integer $v_s + u_s$ by $\chi(c)$, and $(v_s, u_s)$ by $\psi(c)$.

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

**Require:** $P$ and $2P$
**Ensure:** $Q = \chi(c)P$
1: $(U_1, U_2) \leftarrow (P, 2P)$
2: **for** $i = 1 \ldots \text{length}(c)$ **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: **return** $Q = U_1 + U_2$

We will often denote the sequence $(c_1, ..., c_s)$ by $c_1 c_2 \ldots c_s$ for convenience. Let $r$ and $s$ be two integers, we will denote by $cc'$ the element of $\mathcal{M}_{r+s}$ obtained from the concatenation of $c \in \mathcal{M}_r$ and $c' \in \mathcal{M}_s$, so that, for $n > 0$, $c^n$ is a word of $\mathcal{M}_{nr}$.

*Example 1* Let us consider the EAC $c = 00011$ of $\mathcal{M}_5$. It is related to the following sequence of ordered pairs of integers: $(1, 2) \to (2, 3) \to (3, 5) \to (5, 8) \to (5, 13) \to (5, 18)$. So it computes the integer $\chi(c) = 23$, and the couple $\psi(c) = (5, 18)$. Note that $\chi(00011) = \chi(11000) = 23$, so $\chi$ is not injective. Actually, the function $\chi$, even restricted to some $\mathcal{M}_s$ for $s \in \mathbb{N}^*$, is never injective as we can prove that for all $c \in \mathcal{M}_s$ we have $\chi(c_1 \ldots c_s) = \chi(c_s \ldots c_1)$.

Small and big steps have an easy interpretation in terms of linear algebra.

**Definition 2** Let $S_0$ and $S_1$ be the matrices corresponding to the linear maps $(v, u) \mapsto (u, u + v)$ (big step) and $(v, u) \mapsto (v, u + v)$ (small step), namely

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

For $c = (c_1, \ldots, c_s) \in \mathcal{M}_s$, we have the equalities:

$$\psi(c) = (1, 2) \prod_{i=1}^{s} S_{c_i},$$

and

$$\chi(c) = (1, 2) \prod_{i=1}^{s} S_{c_i} \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

A remarkable case is that of the EAC involving big steps only. It corresponds to the sequence of pairs of consecutive Fibonacci numbers $F_n$ defined by $F_0 = 0$, $F_1 = 1$ and $\forall n \in \mathbb{N}$, $F_{n+2} = F_n + F_{n+1}$. Indeed,

$S_0^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix}$, $\psi(0^n) = (F_{n+2}, F_{n+3})$ and $\chi(0^n) = F_{n+4}$.

There are no known regular and efficient methods to find short EAC computing a fixed integer $k$. This is the reason why it was suggested in [21] to randomly generate them from well fitted subsets $\mathcal{S}$, such that the restriction of $\chi$ to $\mathcal{S}$ be injective. For example, Proposition 3 in [21] states that for $(c, c') \in \mathcal{M}_n^2$, the equality $\chi(0^n c) = \chi(0^n c')$ implies $c = c'$. In other words, starting from $(v_0, u_0) = (F_{n+2}, F_{n+3})$ (instead of $(1, 2)$), one obtains $2^n$ different integers when computing $\chi(c)$ for all $c \in \mathcal{M}_n$.

## 3 Euclidean addition chains computing different points

**Definition 3** Let $(a, b) \in \mathbb{N}^2$, $s \in \mathbb{N}^*$ and $c \in \mathcal{M}_s$. We define:

$$\psi_{a,b}(c) = (a, b) \prod_{i=1}^{s} S_{c_i},$$

and

$$\chi_{a,b}(c) = (a, b) \prod_{i=1}^{s} S_{c_i} \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

The case $(a, b) = (1, 2)$ corresponds to Definition 1 as for all $c \in \mathcal{M}_s$ we have $\psi_{1,2}(c) = \psi(c)$ and $\chi_{1,2}(c) = \chi(c)$. Notice that for $c \in \mathcal{M}_s$, the integer $\chi_{a,b}(c)$ is the integer computed as the sum of the two components of the vector obtained from the EAC $c$ when starting from $(a, b)$. In other words, $\chi_{a,b}(c).P$ is the point obtained when applying Algorithm 1 starting from $(aP, bP)$ rather than from $(P, 2P)$. We will need the following Proposition for the two injectivity results presented in this note.

**Proposition 1** *Let $s \in \mathbb{N}^*$. The map $\mu : \mathcal{M}_s \to \mathbb{N}^2$ defined by $\mu(c) = \prod_{i=1}^{s} S_{c_i} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ is injective.*

*Proof.* It is sufficient to prove that $\mu(c) = \mu(c')$ implies $c_1 = c'_1$. Indeed, one could thus conclude left-multiplying by $S_{c_1}^{-1}$ and using induction. To prove this claim, first notice that for all $c \in \mathcal{M}_s$ both components of the vector $\mu(c)$ are positive. Then remark that for any couple of integers $(x, y)$ we have $S_0 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ x+y \end{pmatrix}$ and $S_1 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x+y \\ y \end{pmatrix}$. So, if $\mu(c) = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$, we have $c_1 = 0$ if and only if $\beta > \alpha$. □

**Proposition 2** *Let $n$, $a$ and $b$ be three positive integers such that $a$ and $b$ are co-prime and such that $a > F_{n+2}$ or $b > F_{n+2}$. Then, for all $(c, c') \in \mathcal{M}_n^2$ we have $\chi_{a,b}(c) = \chi_{a,b}(c')$ if and only if $c = c'$.*

To prove this proposition we will make use of the following lemma which follows by an easy induction.

**Lemma 1** *Let $n$ be a non-negative integer, $c \in \mathcal{M}_n$ and $\mu(c) = \begin{pmatrix} x \\ y \end{pmatrix}$. Then*

$$\left(x \leq F_{n+2} \text{ and } y \leq F_{n+1}\right) \text{ or } \left(x \leq F_{n+1} \text{ and } y \leq F_{n+2}\right).$$

As $S_0^n \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix}$ and $S_1 S_0^{n-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix}$ the bound is sharp.

*Proof of Proposition 2.* Let $(c, c') \in \mathcal{M}_n^2$ such that $\chi_{a,b}(c) = \chi_{a,b}(c')$. By definition,

$$\chi_{a,b}(c) = (a, b)\mu(c) \text{ and } \chi_{a,b}(c') = (a, b)\mu(c'). \quad (1)$$

Let us set $\begin{pmatrix} x \\ y \end{pmatrix} = \mu(c)$ and $\begin{pmatrix} x' \\ y' \end{pmatrix} = \mu(c')$, so we have $a(x - x') = b(y' - y)$. Since $a$ and $b$ are co-prime, Gauss lemma implies that $a \mid y' - y$ and $b \mid x - x'$. From Lemma 1 , we have that $|y' - y| \leq F_{n+2}$, thus if $a > F_{n+2}$ we deduce that $y' - y = 0$ and therefore $x - x' = 0$. In the case where $b > F_{n+2}$ we obtain in the same way that $(x, y) = (x', y')$. In both cases it enables to prove that $c = c'$, using Proposition 1. □

*Example 2* With $a = F_{n+2}$ and $b = F_{n+3}$ both conditions of Proposition 2 are satisfied. We recover the case of Proposition 3 in [21], as $\psi_{1,2}(0^n) = (F_{n+2}, F_{n+3})$.

**Corollary 1** *Let $E$ be an elliptic curve and a point $P \in E$ of order $N$. Let $n$, $a$ and $b$ be three positive integers such that*
*- $a$ and $b$ are co-prime,*
*- $a > F_{n+2}$ or $b > F_{n+2}$,*
*- $aF_{n+1} + bF_{n+2} < N$ and $aF_{n+2} + bF_{n+1} < N$.*
*Then the $2^n$ chains $c \in \mathcal{M}_n$ compute $2^n$ different points when applying Algorithm 1 starting from $(aP, bP)$ rather than from $(P, 2P)$.*

*Proof.* Let us consider two different elements $c$ and $c'$ of $\mathcal{M}_n$ as well as the two points $\chi_{a,b}(c)P$ and $\chi_{a,b}(c')P$ in $E$ obtained from Algorithm 1. These two points are the same if and only if $\chi_{a,b}(c)$ is congruent to $\chi_{a,b}(c')$ modulo $N$. The precedent proposition and the first two conditions of the corollary ensure that $\chi_{a,b}(c) \neq \chi_{a,b}(c')$. Since $\chi_{a,b}(c) = (a, b)\mu(c)$ and $\chi_{a,b}(c') = (a, b)\mu(c')$, Lemma 1 and the third condition of the corollary imply that $\chi_{a,b}(c) < N$ and $\chi_{a,b}(c') < N$. Thus $\chi_{a,b}(c)P \neq \chi_{a,b}(c')P$ in $E$. □

*Example 3* Let $E$ be an elliptic curve, and $P$ a point of order $N > F_{2n+4}$. We have $F_{n+2}F_{n+1} + F_{n+3}F_{n+2} = F_{n+2}^2 + F_{n+1}^2 = F_{2n+4}$, so starting from $(F_{n+2}P, F_{n+3}P)$ and applying Algorithm 1 with the $2^n$ chains of $\mathcal{M}_n$ enable us to compute $2^n$ different points of $E$. It corresponds to Method 1 described in [21].

*Example 4* Another possibility is to start from $(P, bP)$, where $b > F_{n+2}$ and the order of the point $P$ is greater than $F_{n+1}+bF_{n+2}$. It requires to precompute the points $P$ and $bP$ with the same $Z$-coordinate unless $bP$ can be efficiently computed on the fly.

## 4 An EAC-based scalar multiplication algorithm for curves with an efficient endomorphism

The first method proposed in [21] requires to start from a pre-computed couple of points $(F_{n+2}P, F_{n+3}P)$. Results from the previous section show that it can be extended to any pair of points $(aP, bP)$ when $a$ and $b$ satisfy the hypotheses of Corollary 1. Now our concern is to adapt these methods to the variable-base scalar multiplication case. Example 4 gives food for thought: in this case we just need $P$ and $bP$. If the curve is endowed by an endomorphism $\phi$, we can obtain $bP = \phi(P)$ without precomputation. However the integer $b$ given in such a way has no reason to verify the hypotheses of Corollary 1. Fortunately, we prove in this section injectivity results when starting from $(P, \phi(P))$.

From now on, we will consider the context of an elliptic curve $E$ endowed with a non trivial endomorphism $\phi$ defined over $\mathbb{F}_p$. We follow the notation adopted by [11]: we fix $P \in E$ a point of prime order $N$ such that $\#(E)/N \leq 4$, and $X^2 + rX + s$ the characteristic polynomial of $\phi$. We will denote by $\lambda$ the unique element of $[0, N-1]$ such that $\phi(P) = \lambda P$. The following result is established in section 2.1 of [11] and in Lemma 6 of [27].

**Proposition 3** *Let $(k_1, k_2) \in \mathbb{Z}^2 \setminus \{(0,0)\}$. If $k_1 + k_2\lambda \equiv 0 \mod N$ then*

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

Combined with Proposition 1 it enables us to prove the following injectivity result.

**Proposition 4** *Under the assumptions above and 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 starting from $(P, \phi(P))$ rather than from $(P, 2P)$.*

*Proof.* Starting from $(P, \phi(P))$ and applying Algorithm 1 with an EAC $c \in \mathcal{M}_n$, one computes $k_1P + k_2\lambda P$, where $\binom{k_1}{k_2} = \prod_{i=1}^{n} S_{c_i}\binom{1}{1}$, that is $\binom{k_1}{k_2} = \mu(c)$. Let $(c, c') \in \mathcal{M}_n^2$ such that $c$ and $c'$ compute the same point starting from $(P, \phi(P))$. Therefore

$$k_1 + k_2\lambda \equiv k_1' + k_2'\lambda \ (N),$$

where $\binom{k_1'}{k_2'} = \mu(c')$. We deduce

$$(k_1 - k_1') + (k_2 - k_2')\lambda \equiv 0 \ (N).$$

But we know that both components of $\mu(c)$ and $\mu(c')$ are less or equal to $F_{n+2}$, we thus have

$$|k_i - k_i'| < \sqrt{\frac{N}{1+|r|+s}} \qquad \text{for} \quad i \in \{1,2\}.$$

Use Proposition 3 to obtain $k_i = k_i'$ for $i \in \{1,2\}$, and Proposition 1 to conclude $c = c'$. □

Based on this result, we propose an alternative way to the classical cryptographic primitive which, starting from a point $P$, maps a random $n$-bit integer $k$ to a random point in the group $< P >$. First randomly generate an EAC $c \in \mathcal{M}_n$. Then, starting from the couple $(P, Q) = (P, \phi(P))$, apply the ZADD addition procedure to obtain, whether the current bit of $c$ is 0 or 1, a new ordered pair of points $(Q, P+Q)$, or $(P, P+Q)$ (see Algorithm 2). Notice that Algorithm 2 uses a slightly different version of ZADD called ZADDb (see Algorithm 3 in the Appendix B). In this version, for each iteration, the coordinates of the two starting points $P$ and $Q$ are used in order to store some intermediate results and are then replaced by the coordinates of the new current couple of points $(P, P+Q)$ or $(Q, P+Q)$. The algorithm takes the current bit of the addition chain as a parameter.

**Algorithm 2** PointFromEAC(EAC $c$)

**Require:** $P(X, Y, 1)$ and $Q = \phi(P) = (X', Y', 1)$
**Ensure:** Update $Q$ with the point computed from $(P, \phi(P))$ and the Euclidean addition chain $c$.

```
for i = 1 ... length(c) do
    ZADDb(c_i)
end for
ZADDb(1)
return Q
```

This way, we obtain a method which maps a random EAC chain $c$ to a point. From a practical point of view, in order to guarantee that we compute $2^n$ distinct points, it is sufficient to satisfy the inequality of Proposition 4. As $F_n = \frac{\gamma^n - \overline{\gamma}^n}{\sqrt{5}}$ where $\gamma = \frac{1+\sqrt{5}}{2}$ and $\overline{\gamma} = \frac{1-\sqrt{5}}{2}$, it is sufficient to choose $N > \gamma^{2n+4}\frac{1+|r|+s}{5}$. The size of the right hand side is equivalent to $2n\log_2(\gamma)$, which is between $1.388n$ and $1.389n$. It amounts to choosing a larger base field, as the size of $E(\mathbb{F}_p)$ is close to $p$ by Hasse-Weil bounds. For convenience, we sum up in Table 1 the size of the field necessary to guarantee the injectivity.

| Security level | 96 | 128 | 192 |
|---|---|---|---|
| Field size | 269 | 358 | 536 |

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

## 5 Implementation and performances (Weierstrass model)

In this section we analyze the computational cost of our method in comparison to the GLV method when using the classical Weierstrass model. We also propose various implementations and we consider the specific context of mobile device to illustrate the relevance of our approach. All our source codes and collected results are available on GitHub:
`https://github.com/eacElliptic`.
The detailed characteristics of the two platforms we used (an Android smartphone and an x64 based computer), as well as the various auxiliary tools are listed in Appendix A.

### 5.1 Theoretical cost comparisons

Let us define $M_t$ as the cost of a modular multiplication of two $t$-bit integers modulo a $t$-bit prime number and $S_t$ that of a squaring. Our scalar multiplication scheme using an EAC of length $l$ consists of $l$ calls to the ZADD procedure ($5M_t + 2S_t$) plus the initial computation of $\phi(P)$. The cost of this computation will not be taken into account neither for GLV nor for our method.

**Important Remark :** From the previous section, in order to guarantee a $\ell/2$-bit security level, $t$ must be chosen close to $1.4\ell$ and such that there exists a curve over a field of this size with an efficient endomorphism.

If we omit the computational cost of finding a suitable decomposition for the scalar $k$ in the form $k_1 + k_2\lambda$, the standard GLV method requires the computation of the points $\phi(P), \phi(P) + P$ and $\phi(P) - P$, the recoding of $k_1$ and $k_2$ in joint sparse form and finally (for a $\ell/2$ bit security level), $\ell/2$ point doublings and on average $\ell/4$ point additions. Each doubling has a cost of $2M_\ell + 5S_\ell$ and each addition (provided that the stored points are in affine coordinates) costs $7M_\ell + 4S_\ell$. However, this method is not secure against Simple Side Channel analysis, that is why we also consider in our comparisons a secure version of the GLV method: GLV-SAC [12]. In that case one point doubling and one point addition is performed for each scalar bit but only the point $\phi(P)+P$ needs to be stored. We summarize the different costs in Table 2. The costs of the classical and secure versions of GLV directly depend on the size of the integer $k$. The cost of our method depends on the length of the EAC used in Algorithm 2. We provide numbers for some specific security levels in Table 3. In any case, the standard GLV method should be faster than the EAC approach but in the case of SPA resistant methods, we can expect our method to be competitive.

Indeed, let us consider the context where multiplication and squaring have the same cost. From Table 2, to obtain a $\ell'$-bit security level, the EAC scalar multiplication algorithm needs $14\ell' + 7$ field multiplications over $t$-bit integers (where $t$ is greater than $2.8\ell'$). The same computation involves $18\ell'$ field multiplications over $2\ell'$-bit integers for the protected version of GLV. Our method should be efficient as soon as

$$M_t < \frac{18\ell'}{14\ell'} M_{2\ell'} \,. \tag{2}$$

Hence , if $M_t < 1.29M_{2\ell'}$, we have an efficient alternative to the GLV protected method.

### 5.2 From theory to practice

From a theoretical point of view, when $t$ is about $2.8\ell'$, the ratio $M_t/M_{2\ell'}$ should be closed to $(1.4)^2$ since the

| Method | field operations |
|---|---|
| EAC | $(\ell + 1)(5M_t + 2S_t)$ |
| W-GLV | $\ell/2 \times (5.5M_\ell + 7S_\ell)$ |
| W-GLV-SAC | $\ell/2 \times (9M_\ell + 9S_\ell)$ |

Table 2: Cost analysis of different scalar multiplication methods for a $\ell/2$-bit security level.

| Method | 96-bit security |
|---|---|
| EAC | $965M_{269} + 386S_{269}$ |
| W-GLV | $528M_{192} + 672S_{192}$ |
| W-GLV-SAC | $864M_{192} + 864S_{192}$ |
| Method | 128-bit security |
| EAC | $1285M_{358} + 514S_{358}$ |
| W-GLV | $704M_{256} + 896S_{256}$ |
| W-GLV-SAC | $1152M_{256} + 1152S_{256}$ |
| Method | 192-bit security |
| EAC | $1925M_{536} + 770S_{536}$ |
| W-GLV | $1056M_{384} + 1344S_{384}$ |
| W-GLV-SAC | $1728M_{384} + 1728S_{384}$ |

Table 3: Theoretical cost of scalar multiplication for a given security level.

classical multiplication algorithm is quadratic. Hence, we do not expect the condition (2) to be verified generally. However, we would like to convince the reader that there exist some real-life contexts in which (2) holds. The aim of this part is to show that it is the case when using general purpose multi-precision libraries (whereas the manipulated data can be stored in few 64-bit words, as already noticed in [13]). There do exist many real-life cryptographic contexts where such libraries are used:

- the popular OpenSSL cryptographic software library relies on the BIGNUM library,
- the secure communication library GnuTLS included in the Synology Diskstation operating system relies on the GNU Multiple Precision arithmetic library,
- the Spongy Castle cryptographic library provided in the Android operating system relies on the Java BigInteger library.

From a practical point of view, the way the arbitrary precision library manages large integers as well as the real amount of memory used to store such integers must be taken into account. Generally speaking, each multiprecision library provides a multiplication procedure which is twofold: it executes the actual computation of the multiplication and also makes some extra operations needed to manage the large integers involved in this computation. So, let us denote by $\tilde{M}_t$ the cost of such a procedure for two $t$-bit integers. Our method will be efficient as soon as

$$\tilde{M}_t < 1.29\tilde{M}_{2\ell'} \,. \tag{3}$$

To identify some contexts where this inequality holds, we have made several benchmarks for modular multiplications over integers of size 192, 269 ($\simeq 192 \times 1.4$), 256, 358 ($\simeq 256 \times 1.4$), 384, 538 ($\simeq 384 \times 1.4$). For these benchmarks we have used the BigInteger Java library provided in the Android Software Development Kit, the BigInteger Java library provided by Oracle in Java SE for x64 platforms and the GNU Multiple Precision arithmetic library. Table 4 sums up the various execution time ratios for $2^{21}$ modular multiplications between $t$-bit integers and $1.4t$-bit integers. Favorable cases for our method appear in boldface. The ratio between 538-bit integers and 384-bit integers is about 1.9 for the multiplication procedure on our Android platform. This is due to the fact that the BigInteger library differently manages integers whose bit-length is less than 512 and integers whose bit-length is greater or equal than 512.

*Remark 1* Notice that for Gnu MP these experimental results should be carefully considered. Indeed, we are near from the 1.29 bound of relation (3). If we only consider the multiplication procedure, the ratio is far enough from 1.29 for $t \in \{192, 256\}$. The problem comes from the additional cost of the modular reduction. Depending on the architecture and the processor used, it should happen that we obtained a ratio for the modular multiplication greater than 1.29 (for $t \in \{192, 256\}$) because of the modular reduction.

| $t$ (bits) | $\simeq 1.4t$ | $\tau_\times$ | $\tau_\%$ | $\tau_*$ |
|---|---|---|---|---|
| 192 | 269 | 1,04095 | 1,05315 | **1,04927** |
| 256 | 358 | 1,00435 | 1,08456 | **1,05934** |
| 384 | 538 | 1,91747 | 1,12702 | 1,36432 |
| Big Integer Android | | | | |
| $t$ (bits) | $\simeq 1.4t$ | $\tau_\times$ | $\tau_\%$ | $\tau_*$ |
| 192 | 269 | 1,75618 | 1,64688 | 1,66882 |
| 256 | 358 | 1,83208 | 1,80546 | 1,81148 |
| 384 | 538 | 1,75453 | 1,89138 | 1,85883 |
| Big Integer Java SE | | | | |
| $t$ (bits) | $\simeq 1.4t$ | $\tau_\times$ | $\tau_\%$ | $\tau_*$ |
| 192 | 269 | 1.15566 | 1.32298 | **1.25854** |
| 256 | 358 | 1.17041 | 1.32226 | **1.26592** |
| 384 | 538 | 1.37065 | 1.55186 | 1.48633 |
| Gnu MP | | | | |

Table 4: Execution time ratio between operations over $1.4t$-bit integers and $t$-bit integers. $\tau_\times$ : ratio for multiplication over integers, $\tau_\%$: ratio for modular reduction, $\tau_*$ : ratio for modular multiplication.

*Remark 2* We have chosen to consider the Android platform because it is widely used in everyday connected objects like smartphones, tablet computers, GPS, car PC, and so on... More generally, there are several Java based platforms and this technology is considered by most of the developers as a de facto standard for developing embedded applications.

Since 2014, the Android operating system includes the Spongy Castle library, a cut-down version of Bouncy Castle Java library. This latter provides a lightweight cryptography API for Java. Bouncy Castle and Spongy castle both use the BigInteger Java library to operate on large integers. This library is also a part of the Java SE provided by Oracle but it differs from the version provided in Android when concerning low-level arithmetic computations. The former is written in pure Java while the latter makes calls to the BIGNUM library written in C and used in OpenSSL.

None of the libraries we used provide a specific method for squaring, so we fit the hypothesis made in the previous subsection about the identical cost of multiplication and squaring.

*5.2.1 Collected results*

To reach the same level of security as the classical GLV method, the conclusion of section 3 suggests that we have to work with integers which are about 1.4 times larger. As shown in Table 4, when using the above mentioned libraries a modular multiplication between $1.4t$-bit integers and $t$-bit integers is:

- about 1.05 times slower on Android for $t \in \{192, 256\}$,
- about 1.26 times slower on an x64 platform with GnuMP for $t \in \{192, 256\}$,
- at least 1.7 times slower on an x64 platform with Java SE for $t \in \{192, 256, 384\}$.

Let us briefly explain these results. When using multiple precision libraries, two factors determine the cost of a modular multiplication: arithmetic (whose cost grows quadratically) and memory management (whose cost grows linearly). In order to understand the total cost of such an operation, we have to consider two cases: both parts are implemented using the same language or each of them is programmed in a specific one.

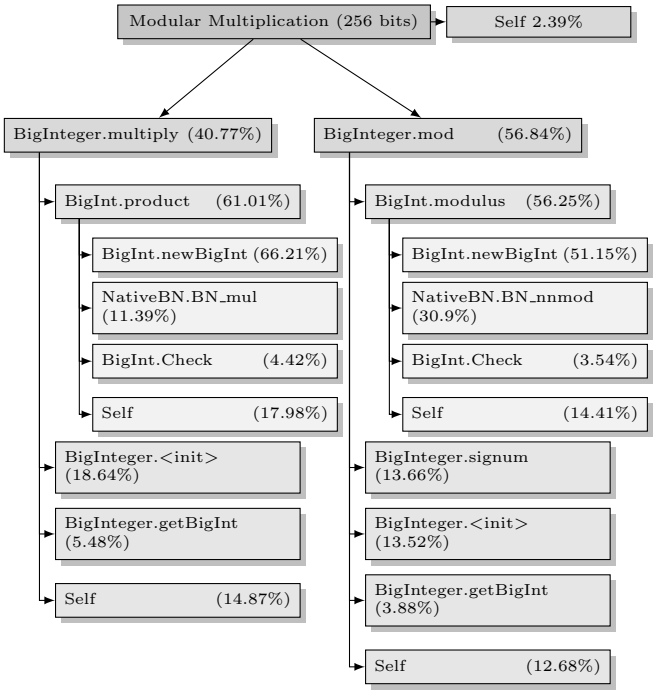
In the first case, the cost strongly depends on the relative cost of arithmetic and memory allocation in the given language. In the second case, the relative speed of the programming language used may have more impact on the overall performances than the cost of the operations themselves. For instance, in the BigInteger Java library provided by Android, arithmetic is performed using a C library (BIGNUM). Even though, the latter is supposed to be quadratic, the fact that Java is much more slower than C makes in this case the memory management be the costliest operation. With regards to the Big Integer Java library provided by Oracle for an x64 platform, arithmetic and memory are both managed in Java. For the size of the manipulated data, memory management performs better than multiplication. For Gnu MP, arithmetic performs better than memory management for the key sizes used in elliptic curve cryptography.

As a concrete illustration, we have used the profiler included in the Android software development kit in order to obtain the *anatomy* of a modular multiplication between two 256-bit integers using the BigInteger Java library (see Figure 1). The code we have developed computes $2^{17}$ products of two random 256-bit integers modulo a random 256-bit prime integer. To compute such a product, our Java method makes a call to the *multiply* and *mod* methods of the BigInteger library which in turn invoke other internal methods. For each of them, the ratio of the execution time of each call is given as well as the ratio of the execution time spent to execute the instructions of the method itself (denoted as "Self" in the figure).

It turns out that the actual product (*NativeBN.BN_mul*) and modular (*NativeBN.BN_nnmod*) operations represent respectively about 2.8% and 9.9% of the whole process. The rest of the time is spent between memory allocation (*BigInt.newBigInt*) and the management of the Java objects used to store large integers (see Figure 2 where we have isolated the time spent for allocation and actual execution from the time spent for other treatments). When the size of the integers is between 192 bits and 384 bits, the time spent for the actual computation of a modular multiplication takes at most 14% of the total execution time. This grows up to at most 20% for 512 to 717-bit integers. Hence for typical cryptographic key sizes, most of the time is spent between memory allocation and management of the Java objects involved in the computation process. From our experimental results, this time is almost identical for $t$-bit integers and $1.4t$-bit integers (see Table 5). This explains the important difference between the theoretical ratio of $(1.4)^2$ and the observed one.

Using the GNU C profiler and the Java Netbeans profiler, we performed a similar study to obtain the anatomy of a modular multiplication for an x64 platform (see Figures 3 and 4 in Appendix C). It turns out that for a Java implementation, we cannot expect to obtain

Fig. 1: Anatomy of a Java method computing a modular multiplication over 256-bit integers using Big Integer Java library provided with Android operating system (obtained from Android SDK profiler).



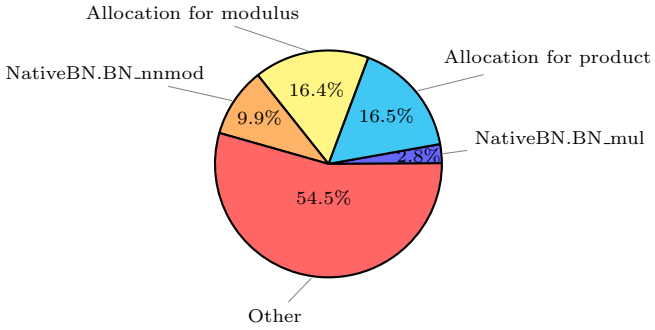



Fig. 2: Distribution of the execution time of various operations involved in the modular multiplication of 256-bit integers on an Android platform.

| $t$ (bits) | $\simeq 1.4t$ | Execution time ratio for memory and object management |
|---|---|---|
| 192 | 269 | 1,00564 |
| 256 | 358 | 0,99838 |
| 384 | 538 | 1,00485 |

Table 5: Execution time ratio for memory allocation and management between $t$-bit BigInteger objects and $1.4t$-bit BigInteger objects.

| Method/Platform | Android |
|---|---|
| EAC/W-GLV (without encoding) | 1.03 |
| EAC/W-GLV (with encoding) | **0.98** |
| EAC/W-GLV-SAC (without enc.) | **0.75** |
| EAC/W-GLV-SAC (with encoding) | **0.73** |

| Method/Platform | Gnu MP, x64 |
|---|---|
| EAC/W-GLV (without encoding) | 1.38 |
| EAC/W-GLV (with encoding) | 1.32 |
| EAC/W-GLV-SAC (without enc.) | **0.98** |
| EAC/W-GLV-SAC (with encoding) | **0.96** |

| Method/Platform | Android ($x$-only) |
|---|---|
| EAC/W-GLV (without encoding) | **0.91** |
| EAC/W-GLV (with encoding) | **0.87** |
| EAC/W-GLV-SAC (without enc.) | **0.69** |
| EAC/W-GLV-SAC (with encoding) | **0.67** |

| Method/Platform | Gnu MP, x64 ($x$-only) |
|---|---|
| EAC/W-GLV (without encoding) | 1.23 |
| EAC/W-GLV (with encoding) | 1.18 |
| EAC/W-GLV-SAC (without enc.) | **0.88** |
| EAC/W-GLV-SAC (with encoding) | **0.86** |

Table 6: Execution time ratio between EAC and GLV schemes for a 128-bit level of security.

a competitive algorithm. Indeed, the actual computation of the multiplication is written in pure Java and its execution time is not a negligible part of the `multiply` procedure.

To sum up, the results of Table 4 show that there exist some practical contexts where the execution time of a modular multiplication between integers used in our EAC scheme satisfies the condition (3).

### *5.2.2 Practical implementations*

To illustrate this purpose, we have implemented on an Android device and on an x64 platform (using Gnu MP) the classical GLV method, the protected version of GLV and our method.

First, notice that the EAC scalar multiplication algorithm needs only few lines of code (see Algorithm 2 and Algorithm 3 in Appendix B). Jacobian coordinates are used to represent all points and only the ZADD addition formula has to be implemented. For GLV and GLV-SAC two initial steps have to be implemented: a precomputation step which splits $k$ into $k_1$ and $k_2$ (Algorithms 4 and 5, Appendix B) and an encoding process to put $k_1$ and $k_2$ in JSF form [33] or SAC form (Algorithm 8, Appendix B). Moreover, we need to implement addition formulas for mixed affine-Jacobian coordinates, as well as doubling formulas.

Although the classical GLV version is included in Spongy castle, we decided to make our own implementation. This way, we aim to make a fair comparison between our method, the protected version of GLV and the classical one. Indeed, Spongy Castle library uses generic Java objects and includes several tests so that the end-user can make his choice between different kinds of curves and scalar multiplication algorithms. This will certainly slow down the execution time as compared to a specific version dealing only with the GLV method.

We have chosen to implement GLV (Algorithm 6, Appendix B) and GLV-SAC (Algorithm 7, Appendix B) for the curve $E(\mathbb{F}_p) : y^2 = x^3 + 5$ with $p = 2^{256} - 1539$, and our EAC scheme for the curve $E(\mathbb{F}_p) : y^2 = x^3 + 17$ with $p = 2^{358} - 36855$. These two sets of parameters guarantee a 128-bit security level (see Table 1). The corresponding endomorphism $\phi$ satisfies $\phi^2 + \phi + 1 = 0$. It corresponds to the map $(x, y) \mapsto (\beta x, y)$ where $\beta$ is an element of order 3.

In order to compare these methods, we have recorded the average execution time of $2^{17}$ scalar multiplications. Table 6 gives the corresponding ratios. The favorable cases for our method appear in boldface. Notice that for the GLV methods, the computation of $kP$ (for various points $P$) requires to decompose $k$ into two smallest integers $k_1$ and $k_2$. This encoding process is often over-

looked in the computational cost of $kP$. However, in some contexts, $k$ and $P$ can both change and thus this encoding process can slow down the whole GLV execution. The encoding process for GLV is done by Algorithms 4 and 5 (Appendix B) as described in [31]. The running time of this decomposition is better than those of the original encoding process proposed by Gallant, Lambert and Vanstone. For GLV, the integers $k_1$ and $k_2$ must then be put into Joint Sparse Form [33]. For GLV-SAC, $k_1$ and $k_2$ must be post-processed by Algorithm 8 (Appendix B, adapted from [12] to fit in our context).

Instead of randomly choosing the integer $k$, another way is to randomly generate $k_1$ and $k_2$ as suggested and justified in [9]. Notice that in this case the *recomposition* of $k$ from $k_1$ and $k_2$ is no more injective. In Table 6, we made comparisons with two versions of GLV: one with the encoding of $k$, and the other one without encoding.

It turns out that on an Android platform, the EAC scalar multiplication algorithm performs well as compared to the classical GLV algorithm (only 3% slower) and can even be faster (2%) in the context where $k$ and $P$ both vary. The most significant gain is obtained when taking into account the resistance to side channel attacks since our scheme is at least 25% faster than the protected version of GLV. For the x64 platform, our execution time is 2% faster than the secure version of GLV (this result should be carefully considered as explained in Remark 1). We obtain a more significant gain if we target $x$-coordinate-only system (see Section 9 and [9]).

*Remark 3* We insist on the fact that once we have chosen a general multi-precision library, our constraint is to use the primitives offered by this latter. Hence, we do not consider any optimized modular reduction.

Let us focus on the classical multiplication operation over integers. For 96-bit and 128-bit security levels, Table 4 (column $\tau_\times$) shows that the EAC method should always perform better than GLV-SAC on an Android platform (or an x64 platform using GNU MP). Hence, if we consider both for GLV-SAC and our algorithm some specific primes $p$ (like in our examples), using fast modulo reduction techniques will certainly optimize numbers given in the column $\tau_\%$. Thereby, compared to GLV-SAC, the performances of the EAC algorithm should be better.

*Remark 4* The fact that our method is only 3% slower than the classical GLV method (without encoding) may seem curious. The results of Table 3 show that there are 1.12 as many multiplications in the EAC based algorithm than in the GLV method. Since, from Table 4, a modular multiplication between two 358-bit integers is about 1.06 times slower than a multiplication between two 256-bit integers, this should lead us to an execution time ratio of $1.12 \times 1.06 \simeq 1.187$ rather than 1.03. Now, let us isolate in our experimental results the time spent for modular multiplications during the scalar multiplication algorithm. We observe a ratio of about 1.19. This result confirms the theoretical analysis of subsection 5.1. This gives us the opportunity to emphasize the need for caution when only considering the cost of the modular multiplication in order to state the complexity of a scalar multiplication algorithm. Indeed, if we take into account the whole computation process, including modular additions, modular multiplications by constants which appear in point addition formulas and the management of the different auxiliary Java objects used in all these computations, the theoretical ratio of 1.19 boils down to a practical ratio of 1.03. Addition formulas used for GLV [4] involve some field multiplications by a constant. Even if those multiplications can be done using the `BigInteger.shiftleft()` method of the BigInteger library, their cost is non-negligible.

## 6 Implementation and performances (extended twisted Edwards curves model)

Ted127-glv4 [12] and FourQ [10] are the two actual most efficient (and secure) scalar multiplication algorithms. They rely on the four-dimensional GLV algorithm and take advantage of two efficient endomorphisms and the extended twisted Edwards coordinates [22], which offer the fastest known curves addition formulas over large prime characteristic fields. Nevertheless, some precomputed points need to be stored for table lookups. As an example, 512 bytes of memory are needed to store 8 points in Ted127-glv4.
A raw comparison of our approach with these methods is complicated. On the one hand, without a doubt, methods using two endomorphisms are faster, on the other hand our implementation context is quite different (arithmetic over $\mathbb{F}_p$, low memory usage and curves with one endomorphism). However, it seems fair to take advantage of the twisted Edwards coordinates system, to compare our approach to the 2-dimensional version of GLV-SAC, which matches our context.

### 6.1 Theoretical cost comparisons with GLV-SAC

In extended twisted Edwards coordinates, a point is represented by four coordinates $(X, Y, T, Z)$ with $T = XY/Z$. Using Algorithm 7, each point doubling is followed by a point addition. The best performances for

this sequence of operations is obtained by mixing standard twisted Edwards coordinates with extended twisted Edwards coordinates (as explained in [22]). This leads to $10M_\ell + 4S_\ell$ for a $\ell/2$-bit security level (see Table 7 and 8). It appears from this first theoretical approach, that we cannot outperform GLV-SAC when using extended twisted Edwards coordinates.

Once again, experimental results (see Table 9) reveal that we have to carefully take into account every computations in order to evaluate the real cost of a scalar multiplication algorithm. Results from Table 9 have been obtained by implementing GLV-SAC on the twisted Edwards curve $E(\mathbb{F}_p) : -x^2 + y^2 = 1 + x^2y^2$ with $p = 2^{256} - 43443$. The order of the curve is $8.h$ for some prime $h$. The corresponding endomorphism $\phi$ satisfies $\phi^2+1 = 0$. It corresponds to the map $(x, y) \mapsto (\alpha x, 1/y)$ where $\alpha$ is an element of order 4 [25]. Parameters for our EAC scheme are those defined in section 5.2.2. In order to compare these two methods, we have recorded the average execution time of $2^{17}$ scalar multiplications.

Let us explain the results obtained on the Android platform. First, from Table 4, we see that the computational cost of a 358-bit modular multiplication barely exceeds that of a 256-bit one. The ZADDb procedure involves precisely 7 modular multiplications (assuming the same cost for squaring and multiplying operations) and 7 modular additions. The mixed coordinate addition for the twisted Edward curves [22] involves precisely 14 multiplications, 14 additions, 1 multiplication by 2 and on average the computation of a modular opposite:

1. a doubling requires 8 multiplications, 6 additions and a multiplication by the curve constant $a$, which in our case is equal to -1,
2. an addition requires 6 multiplications, 8 additions and two multiplications by 2 [22],
3. half the time the opposite of one of the two affine points $P$ or $P + \phi(P)$ has to be computed.

The multiplication by 2 can be implemented using either the `BigInteger.shiftleft()` method or using the `BigInteger.add()` method. Experimental results show that these two methods have almost the same execution time. Indeed, these two methods allocate a new BigInteger object, and as it has already been discussed in section 5.2.1, the most important part in the execution time of an operation comes from the memory allocation.

Similarly, the modular opposite can be implemented using either the `BigInteger.neg()` method or the `BigInteger.subtract()` method. For the reasons mentioned above, these two operations are almost equivalent.

Table 10 summarizes the cost of those two methods for a $\ell$-bit security level ($D_\ell$ denotes the multiplication by 2 of an $\ell$-bit integer and $N_\ell$ the modular opposite of an $\ell$-bit integer). As soon as computing $\ell(3D_\ell/2 + N_\ell)$ is slower than computing $7M_t + 7A_t$ our approach becomes more efficient. In other words, we have to compare the allocation time of 14 BigInteger objects used to compute $7M_t + 7A_t$, with the allocation time of $5\ell/2$ BigInteger objects used to compute $\ell(3D_\ell/2 + N_\ell)$.

Let $T_{\text{EAC}}$ be the execution time of our EAC method. Considering that the execution time of a 256-bit and a

| Method | field operations |
|---|---|
| EAC | $(\ell + 1)(5M_t + 2S_t)$ |
| TED-GLV-SAC | $\ell/2 \times (10M_\ell + 4S_\ell)$ |

Table 7: Cost analysis of EAC and TED-GLV-SAC for a $\ell/2$-bit security level ($t \simeq 1.4\ell$).

| Method | 96-bit security |
|---|---|
| EAC | $965M_{269} + 386S_{269}$ |
| TED-GLV-SAC | $960M_{192} + 384S_{192}$ |
| **Method** | **128-bit security** |
| EAC | $1285M_{358} + 514S_{358}$ |
| TED-GLV-SAC | $1280M_{256} + 512S_{256}$ |
| **Method** | **192-bit security** |
| EAC | $1925M_{536} + 770S_{536}$ |
| TED-GLV-SAC | $1920M_{384} + 768S_{384}$ |

Table 8: Theoretical cost of scalar multiplication for a given security level.

| Method/Platform | Android |
|---|---|
| EAC/TED-GLV-SAC (without enc.) | **0.95** |
| EAC/TED-GLV-SAC (with enc.) | **0.93** |

| Method/Platform | Gnu MP, x64 |
|---|---|
| EAC/TED-GLV-SAC (without enc.) | 1.41 |
| EAC/TED-GLV-SAC (with enc.) | 1.39 |

| Method/Platform | Android ($x$-only) |
|---|---|
| EAC/TED-GLV-SAC (without enc.) | **0.84** |
| EAC/TED-GLV-SAC (with enc.) | **0.82** |

| Method/Platform | Gnu MP, x64 ($x$-only) |
|---|---|
| EAC/TED-GLV-SAC (without enc.) | 1.21 |
| EAC/TED-GLV-SAC (with enc.) | 1.19 |

Table 9: Execution time ratio between EAC and TED-GLV-SAC schemes for a 128-bit level of security.

| Method | field operations |
|---|---|
| EAC | $(\ell+1)(7M_t+7A_t)$ |
| TED-GLV-SAC | $\ell/2\times(14M_\ell+14A_\ell+3D_\ell+2N_\ell)$ |

Table 10: Cost analysis of EAC and TED-GLV-SAC for a $\ell/2$-bit security level ($t\simeq 1.4\ell$).

358-bit modular multiplication (or addition) is roughthly the same, the computational cost of TED-GLV-SAC should be $T_{\text{EAC}}+\delta(\frac{5\times 256}{2}-14)$ where $\delta$ is the execution time of memory allocation. Those allocations are done through calls to the `BigInt.newBigInt()` method. Experimentation has shown that its execution time is about 50 microseconds which leads to a ratio of

$$\frac{T_{\text{EAC}}}{T_{\text{EAC}}+31300}.$$

Performing $2^{17}$ scalar multiplications, we have obtained an average execution time of 693943 microseconds for $T_{\text{EAC}}$, which gives a ratio of about 0.96, in accordance with the results of row 1 in Table 9. Notice that the same analysis cannot be done for the x64 platform: the difference between the execution times of modular multiplications is too large (see Table 4).

It turns out that on an Android platform, the EAC scalar multiplication algorithm performs well as compared to the 2-dimensional GLV-SAC algorithm even in the context of extended twisted Edwards coordinates. Using the $x$-coordinate-only system makes the comparison even more favorable.

On an x64 platform, on the one hand the $x$-coordinate-only version of EAC is 1.2 times slower than TED-GLV-SAC, but on the other hand TED-GLV-SAC needs 1.4 times more memory (see section 8). Our method achieves a reasonable trade off between execution time and data storage.

## 7 Security

Our algorithm is based on the constant time `ZADDb` procedure making it intrinsically immune to simple side channel attacks like power consumption measurement. Moreover, the procedure is composed of a fixed sequence of instructions (mixing multiplications and subtractions over the prime field) which does not depend on the scalar $k$. This is enough to thwart many instruction cache timing attacks [1,3,35]. Yet, some additional precaution must be taken in order to resist to the attack proposed in [7]. Indeed, considering the table built from the precomputed points, each bit of the key leads to a table lookup to select one of these points. The corresponding index is a scalar digit which can be revealed thanks to a cache timing trace. Since the selected index is linked to the key, the attacker can derive the value of the key. GLV-SAC is vulnerable to this attack. As a possible countermeasure, the author of [24] proposes to transverse the entire table for each table lookup and to use some functions to compute the correct index. That means that the entire table is loaded into the cache. In the `ZADDb` procedure, we have a table of two points and, at each iteration, these two points are used. Consequently, their addresses should always be loaded into the cache. However, as it can be seen in the `ZADDb` procedure, some operations depend on the address of the point that is used. As an example, in line 4, the CPU will either load $X_0$ or $X_1$ in order to perform the multiplication $X_b.C$. An attacker could actually fill the cache with values to observe timing differences depending on where and how many of the operands $X_0$ and $X_1$ are used. In our context, the countermeasure described in [24] can be simplified because we only need to transverse two points. We describe in Algorithm 10 a safe permutation that returns a pair of points $(P,Q)$ or $(Q,P)$ depending on bit $b$. We then adapt the `ZADDb` procedure and our EAC scheme to take into account this safe permutation (see Algorithms 9 and 11). We will see in the next section why this countermeasure cannot be applied as such for TED-GLV-SAC.

## 8 Memory usage

Table 11 sums up (for a 128-bit level of security) the number of registers needed to store the coordinates of the various points involved in the computation of $kP$ as well as the number of auxiliary registers used for the addition and doubling formulas (see Algorithm 3 in Appendix B and [4]).

Notice that for the EAC algorithm, the two initial input points are represented using Jacobian coordinates and share a common coordinate $Z$. Moreover, since the input points are used to store intermediate results as well as the coordinates of the computed point (Algorithm 2, Appendix B), the `ZADDb` procedure can be implemented with only two additional registers.
For GLV (resp. GLV-SAC), in the Weierstrass model, the four (resp. two) input points are in affine coordinates, while intermediate points and the computed point are in Jacobian coordinates. For TED-GLV-SAC, the two input points are in extended affine coordinates $(x,y,xy,1)$, and the intermediate point $Q$ is either represented in extended or standard twisted Edwards coordinates. In both cases, intermediate results cannot be stored into the coordinates of the input points because

Algorithms 6 and 7 in Appendix B use these points at each iteration of their main loop. Four (resp. two) additional registers are needed for the whole computation of GLV and GLV-SAC (resp. TED-GLV-SAC).

| Method | EAC | EAC $x$-only |
|---|---|---|
| Field size | 358 | 358 |
| Registers for points | 5 | 4 |
| Auxiliary registers | 2 | 2 |
| Mem. size (bits) | 2506 | 2148 |
| Mem. size (bytes) | 314 | 269 |
| Mem. size (32-bit) | 79 | 68 |
| Mem. size (64-bit) | 40 | 34 |

| Method | GLV | GLV-SAC | TED-GLV-SAC |
|---|---|---|---|
| Field size | 256 | 256 | 256 |
| Registers for points | 11 | 7 | 10 |
| Auxiliary registers | 4 | 4 | 2 |
| Mem. size (bits) | 3840 | 2816 | 3072 |
| Mem. size (bytes) | 480 | 352 | 384 |
| Mem. size (32-bit) | 120 | 88 | 96 |
| Mem. size (64-bit) | 60 | 44 | 48 |

Table 11: Memory usage in bits/8-bit words/32-bit words/64-bit words for a 128-bit level of security.

*Remark 5* Our countermeasure (see previous section) cannot be applied as such for GLV-SAC and TED-GLV-SAC even though two precomputed points are used. The points $P$ or $P+\phi(P)$ can be safely selected. However, half the time the points $-P$ or $-(P+\phi(P))$ have to be computed, leading to possible leakage. To prevent this, the four points $P$, $P+\phi(P)$, $-P$ and $-(P+\phi(P))$ must be stored and the table lookup must be implemented as suggested in [24].

*Remark 6* A faster version of GLV-SAC for 2-dimensional GLV is described in section 3.2 of [12]. The drawback is that eight points must be stored instead of two. The same remark can be made for the two fastest scalar multiplication algorithms yet, namely Ted127-glv4 and FourQ. Both algorithms each requires 512 bytes to store the precomputed points when ours only requires 224 bytes.

Compared to the methods using at least one endomorphism, the EAC scalar multiplication algorithm is less memory consuming. Hence it seems well suited for memory constrained devices using for example a Java virtual machine. Now, on memory constrained devices where the cryptographic operations are performed by a specific library located in ROM or EEPROM/Flash that directly accesses the RAM , our algorithm will be less efficient because in this context, equation (3) will certainly not be satisfied.

## 9 Other Related works

In [9] the authors use a curve and its quadratic twist (over $\mathbb{F}_{p^2}$) to obtain a fast $x$-coordinate-only scalar multiplication algorithm using 256 bytes of memory for precomputations. Our method can also be derived in an $x$-coordinate-only scheme as described in [28] (see Table 6) and only uses 179 bytes of memory. Moreover, contrary to the method proposed in [9], our algorithm guarantees that for two distinct inputs, two distinct points are output.

An implementation of GLV (standard and secure version) for OpenSSL is discussed in [6]. For a 128-bit security level, the proposed secure version is about 10% slower than the standard GLV version. We cannot make a practical comparison with this proposal because we did not investigate the OpenSSL library. Now, OpenSSL and Android both rely on the BIGNUM C-library for arithmetic over large integers. We can just notice that on Android, the performances of both classical GLV and our method are very close to one another (and 10% faster for an $x$-only-coordinate system). On the other hand, on an x64 platform with Gnu MP, we have an additional cost of about 40% (and almost no additional cost for an $x$-coordinate-only system).

Notice that another way to naturally obtain an $x$-only-coordinate system for our method is to use the work described in [17]. In this paper the authors describe a modified version of the ZADD operation such that all the computations can be done using only affine coordinates. Since our scalar multiplication algorithm is based on the ZADD primitive, we can benefit from this procedure.

## 10 Conclusion

In this work we have given a mathematical context which enables to use Euclidean addition chains on a curve with one endomorphism. Therefore we have proposed an efficient and secure algorithm for scalar multiplication which can be very easily implemented. The proposed method does not compete with the fastest ones on curves endowed with two endomorphisms. However, up to our knowledge, ours appears to be the less memory consuming among all endomorphism based methods. For developers using the standard cryptographic library included in the Android operating system, we provide a competitive algorithm as compared to the well known GLV method using one endomorphism, even when using extended twisted Edward coordinates.

**Acknowledgements:** we would like to thank the referees for their careful reading and their helpful comments.

## References


1. Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Proceedings of CHES 2010*, volume 6225 of *LNCS*, pages 110–124. Springer Berlin Heidelberg, 2010.
2. B. Baldwin, R. R. Goundar, M. Hamilton, and W. P. Marnane. Co-z ECC scalar multiplications for hardware, software and hardware-software co-design on embedded systems. *J. Cryptographic Engineering*, 2(4):221–240, 2012.
3. Naomi Benger, Joop Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit": A small amount of side channel can go a long way. In *Proceedings of CHES 2014*, volume 8731, pages 75–92, New York, NY, USA, 2014. Springer-Verlag New York, Inc.
4. D. J. Bernstein and T. Lange. Explicit-Formulas Database. `https://www.hyperelliptic.org/EFD/`.
5. A. Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45(10):736–739, 1939.
6. Billy Bob Brumley. Faster software for fast endomorphisms. In *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015*, pages 127–140, 2015.
7. Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *Advances in Cryptology – ASIACRYPT 2009: 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan*, pages 667–684. Springer Berlin Heidelberg, 2009.
8. H. Cohen, G. Frey, R. M. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Boca Raton, FL: Chapman & Hall/CRC, 2006.
9. C. Costello, H. Hisil, and B. Smith. Faster compact Diffie–Hellman: endomorphisms on the x-line. In *Advances in Cryptology–EUROCRYPT 2014*, pages 183–200. Springer, 2014.
10. Craig Costello and Patrick Longa. Four$\mathbb{Q}$: four-dimensional decompositions on a $\mathbb{Q}$-curve over the Mersenne prime. In *Advances in cryptology – ASIACRYPT 2015. 21st international conference on the theory and application of cryptology and information security, Auckland, New Zealand, November 29 – December 3, 2015. Proceedings. Part I*, pages 214–235. Berlin: Springer, 2015.
11. F. Sica and M. Ciet and J-J. Quisquater. 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, 2003.
12. A. Faz-Hernández, P. Longa, and A. H. Sánchez. 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, 2015.
13. S. Galbraith, X. Lin, and M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. *Journal of Cryptology*, 24(3):446–469, 2011.
14. S. D. Galbraith, X. Lin, and M. Scott. 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, 2009.
15. R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology — CRYPTO*, volume 2139 of *LNCS*, pages 190–200. Springer, 2001.
16. R. R. Goundar, M. Joye, and A. Miyaji. Co-$Z$ addition formulæ and binary ladders on elliptic curves - (extended abstract). In *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 65–79, 2010.
17. Raveen R. Goundar and Marc Joye. Inversion-free arithmetic on elliptic curves through isomorphisms. *Journal of Cryptographic Engineering*, pages 1–13, 2016.
18. Raveen R. Goundar, Marc Joye, Atsuko Miyaji, Matthieu Rivain, and Alexandre Venelli. Scalar multiplication on Weierstraß elliptic curves from co-z arithmetic. *Journal of Cryptographic Engineering*, 1(2):161–176, 2011.
19. A. Guillevic and S. Ionica. Four-dimensional GLV via the weil restriction. In *Advances in Cryptology - ASIACRYPT 2013*, pages 79–96, 2013.
20. D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.
21. F. Herbaut, P-Y. Liardet, N. Méloni, Y. Téglia, and P. Véron. Random euclidean addition chain generation and its application to point multiplication. In *Progress in Cryptology - INDOCRYPT 2010*, volume 6498 of *LNCS*, pages 238–261. Springer Berlin / Heidelberg, 2010.
22. Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards curves revisited. In *Advances in cryptology – ASIACRYPT 2008. 14th international conference on the theory and application of cryptology and information security, Melbourne, Australia, December 7–11, 2008. Proceedings*, pages 326–343. Berlin: Springer, 2008.
23. M. Hutter, M. Joye, and Y. Sierra. Memory-constrained implementations of elliptic curve cryptography in co-$Z$ coordinate representation. In *Progress in Cryptology - AFRICACRYPT 2011*, pages 170–187, 2011.
24. Emilia Käsper. Fast elliptic curve cryptography in openssl. In *Proceedings of the 2011 International Conference on Financial Cryptography and Data Security*, FC'11, pages 27–39. Springer-Verlag, 2012.
25. Zhe Liu, Husen Wang, Johann Großschädl, Zhi Hu, and Ingrid Verbauwhede. Vlsi implementation of double-base scalar multiplication on a twisted edwards curve with an efficiently computable endomorphism. *IACR Cryptology ePrint Archive*, 2015:421, 2015.
26. Patrick Longa and Ali Miri. *New Composite Operations and Precomputation Scheme for Elliptic Curve Cryptosystems over Prime Fields*, pages 229–247. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
27. Patrick Longa and Francesco Sica. Four-dimensional Gallant–Lambert–Vanstone scalar multiplication. *Journal of Cryptology*, 27(2):248–283, 2014.
28. N. Meloni. New point addition formulae for ECC applications. In *Arithmetic of Finite Fields*, volume 4547 of *LNCS*, pages 189–201. Springer Berlin / Heidelberg, 2007.
29. B. Möller. Improved techniques for fast exponentiation. In Springer Berlin / Heidelberg, editor, *Information Security and Cryptology — ICISC 2002*, volume 2587 of *LNCS*, pages 298–312, 2003.
30. P. L. Montgomery. Evaluating Recurrences of form $x_{m+n} = f(x_m, x_n, x_{m-n})$ via Lucas chains, 1983. Available at ftp.cwi.nl:/pub/pmontgom/Lucas.ps.gz.

31. Y-H. Park, S. Jeong, C. Kim, and J. Lim. An alternate decomposition of an integer for faster point multiplication on certain elliptic curves. In *Public Key Cryptography*, volume 2274 of *LNCS*, pages 323–334. Springer Berlin Heidelberg, 2002.
32. B. Smith. Families of fast elliptic curves from ıı-curves. In *Advances in Cryptology - ASIACRYPT 2013*, pages 61–78, 2013.
33. J. A. Solinas. Low-weight binary representations for pairs of integers. Technical report, University of Waterloo. Department of Combinatorics and Optimization, 2001.
34. E. G. Thurber. On addition chains $l(mn) \leq l(n) - b$ and lower bounds for $c(r)$. *Duke Mathematical Journal*, 40:907–913, 1973.
35. Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

## Appendix A Benchmark Platforms

All our source codes and collected results are available on GitHub:
`https://github.com/eacElliptic`.

The characteristics of the platforms we used for our benchmarks are the following:

- Android platform : Wiko Cik Peax 2 phone, with MediaTek MT6589 CPU (4 -core ARM Cortex-A7, 1.21 GHz), Android Version : 4.1.2, API Level 16.
- Java platform : Intel Core I5-4210U 4-core 1.7Ghz, Broadwell technology, JDK 1.7.0_79, Ubuntu 14.04 LTS.
- C platform : Intel Core I5-4210U 4-core 1.7Ghz, Broadwell technology, gcc 5.2.1, gmp 6.1.0, Ubuntu 14.04 LTS.

From various benchmarks, when the CPU models are exactly the same, it seems that on a given task Broadwell technology is better than Haswell technology by 5 percent to 10 percent or so. The Intel Turbo Boost technology has been disabled on the x64 platform so that the frequency of the processor be constant.

To collect the various execution results we have used the following tools:

- Android platform: the `startMethodTracing` and the `stopMethodTracing` of the `Debug` class to generate trace logs, and the `System.currentTimeMillis()` method to measure execution time,
- Java platform: the profiler provided with Netbeans IDE (v. 8.1) and the `System.currentTimeMillis()` method,
- C platform: the `clock.gettime()` system call with `CLOCK_PROCESS_CPUTIME_ID` option, and `taskset` to bind our running process to only one processor.

## Appendix B Algorithms

**Algorithm 3** ZADDb(bit $b$)

**Require:** $P(X_0, Y_0, Z)$ and $Q(X_1, Y_1, Z)$
**Ensure:** Update $X_0, Y_0, X_1, Y_1$ and $Z$ such that $(X_0, Y_0, Z)$ and $(X_1, Y_1, Z)$ be the representatives of $P$ and $P + Q$ (or $Q$ and $P + Q$ whether $b$ is 1 or 0).

$C \leftarrow X_{1-b} - X_b$
$Z \leftarrow Z.C$
$C \leftarrow C^2$
$W \leftarrow X_b.C$
$X_0 \leftarrow X_{1-b}.C$
$C \leftarrow Y_{1-b} - Y_b$
$X_1 \leftarrow C^2 - W - X_0$
$W \leftarrow X_0 - W$
$Y_0 \leftarrow Y_{1-b}.W$
$W \leftarrow X_0 - X_1$
$Y_1 \leftarrow C.W - Y_1$

**Algorithm 4** Precomputation for GLV

**Require:** $\lambda$ root of $\phi^2 + \phi + 1 = 0 \pmod{\#E(\mathbb{F}_p)}$
**Ensure:** $(a, b)$ is a short vector such that $a + b\lambda \equiv 0 \pmod{\#E(\mathbb{F}_p)}$, $N_\alpha = N_{\mathbb{Z}[\phi]/\mathbb{Z}}(a + b\phi)$.

$u \leftarrow \#E(\mathbb{F}_p)$, $v \leftarrow \lambda$, $q \leftarrow 0$, $y \leftarrow 0$
$x_1 \leftarrow 1$, $y_1 \leftarrow 0$, $x_2 \leftarrow 0$, $y_1 \leftarrow 1$
**while** $u > \sqrt{\#E(\mathbb{F}_p)}$ **do**
    $q \leftarrow \lfloor v/u \rfloor$, $r \leftarrow v - qu$
    $x \leftarrow x_2 - qx_1$, $y \leftarrow y_2 - qy_1$
    $v \leftarrow u$, $u \leftarrow r$
    $x_2 \leftarrow x_1$, $x_1 \leftarrow x$
    $y_2 \leftarrow y_1$, $y_1 \leftarrow y$
**end while**
$b \leftarrow -y$
$a \leftarrow u + y$
$N_\alpha \leftarrow u^2 + b^2 - ub$

**Algorithm 5** Decompose($k$)

**Require:** $a$, $b$, $N_\alpha$
**Ensure:** $k_1$ and $k_2$ satisfy $kP = k_1P + k_2\phi(P)$
$x_1 \leftarrow k(a+b)$, $x_2 \leftarrow -kb$
$y_1 \leftarrow \lfloor x_1/N_\alpha \rfloor$, $y_2 \leftarrow \lfloor x_2/N_\alpha \rfloor$
$k_1 \leftarrow k - (ay_1 - by_2)$, $k_2 \leftarrow -(ay_2 + b_y 1 + by_2)$
$k_1 \leftarrow k_1 + k_2$
**return** $(k_1, k_2)$

**Algorithm 6** PointFromGLV($k$, $PP$)

**Require:** $PP$ is $(P, -P + \phi(P), \phi(P), P + \phi(P))$
**Ensure:** $Q = kP$
$(k_1, k_2) \leftarrow$ Decompose($k$)
$((x_j, \ldots, x_0), (y_j, \ldots, y_0)) \leftarrow$ SJSF($k_1$, $k_2$)
$u \leftarrow x_j + 3.y_j$
$Q \leftarrow (X_{PP[|u|-1]}, \text{sign}(u).Y_{PP[|u|-1]})$
$j \leftarrow j - 1$
**while** $(j \geqslant 0)$ **do**
    $Q \leftarrow 2Q$
    $u \leftarrow x_j + 3.y_j$
    **if** $u \neq 0$ **then**
        $Q \leftarrow Q + (X_{PP[|u|-1]}, \text{sign}(u).Y_{PP[|u|-1]})$
    **end if**
**end while**
**return** $Q$

**Algorithm 7** PointFromSGLV($k$, $PP$)

**Require:** $PP$ is $(P, P + \phi(P))$
**Ensure:** $Q = kP$
$(k_1, k_2) \leftarrow$ Decompose($k$)
**if** $k_1$ is even **then**
    $k_1 \leftarrow k_1 - 1$
**end if**
$((x_j, \ldots, x_0), (y_j, \ldots, y_0)) \leftarrow$ GLV-SAC($k_1$, $k_2$)
$Q \leftarrow (X_{PP[|y_j|]}, \text{sign}(x_j).Y_{PP[|y_j|]})$
$j \leftarrow j - 1$
**while** $(j \geqslant 0)$ **do**
    $Q \leftarrow 2Q$
    $Q \leftarrow Q + (X_{PP[|y_j|]}, \text{sign}(x_j).Y_{PP[|y_j|]})$
**end while**
**if** $k_1$ is even **then**
    $Q \leftarrow Q + (X_{PP[0]}, Y_{PP[0]})$
**end if**
**return** $Q$

**Algorithm 8** GLV-SAC($k_1$, $k_2$)

**Require:** $k_1 = (k_{\ell-1}^{(1)}, \ldots, k_0^{(1)})$ and $k_2 = (k_{\ell-1}^{(2)}, \ldots, k_0^{(2)})$ are $\ell$-bit positive integers. $k_1$ is odd and $\ell = \#E(\mathbb{F}_p)$.
**Ensure:** Output $(b_{\ell-1}^{(1)}, \ldots, b_0^{(1)})$ and $(b_{\ell-1}^{(2)}, \ldots, b_0^{(2)})$ such that $\forall i$, $b_i^{(1)} \in \{-1, 1\}$ and $b_i^{(2)} \in \{0, b_i^{(1)}\}$.
$b_{\ell-1}^{(1)} \leftarrow 1$
**for** $i = 0, \ldots, \ell - 2$ **do**
    $b_i^{(1)} \leftarrow 2k_{i+1}^{(1)} - 1$
    $b_i^{(2)} \leftarrow b_i^{(1)}.k_0^{(2)}$
    $k_2 \leftarrow \lfloor k_2/2 \rfloor - \lfloor b_i^{(2)}/2 \rfloor$
**end for**
$b_{\ell-1}^{(2)} \leftarrow k_0^{(2)}$
**return** $(b_{\ell-1}^{(1)}, \ldots, b_0^{(1)})$, $(b_{\ell-1}^{(2)}, \ldots, b_0^{(2)})$

**Algorithm 9** ZADDU($P$, $Q$)

**Require:** $P(X_0, Y_0, Z)$ and $Q(X_1, Y_1, Z)$
**Ensure:** Update $X_0, Y_0, X_1, Y_1$ and $Z$ such that $(X_0, Y_0, Z)$ and $(X_1, Y_1, Z)$ be the representatives of $P$ and $P + Q$.
$A \leftarrow X_1 - X_0$
$Z \leftarrow Z.A$
$A \leftarrow A^2$
$X_0 \leftarrow X_0.A$
$A \leftarrow X_1.A$
$Y_1 \leftarrow Y_1 - Y_0$
$B \leftarrow Y_1^2$
$X_1 \leftarrow B - X_0 - A$
$A \leftarrow A - X_0$
$Y_0 \leftarrow Y_0.A$
$B \leftarrow X_0 - X_1$
$Y_1 \leftarrow Y_1.B - Y_0$

**Algorithm 10** SafePerm($P$, $Q$, bit $bit$)

**Require:** $P(X_0, Y_0, Z)$, $Q(X_1, Y_1, Z)$ and $bit$
**Ensure:** Permute safely $P$ and $Q$ if $bit$ is equal to 0.
mask $\leftarrow (bit - 1)$
$X_0 \leftarrow X_0 \oplus X_1$
$X_1 \leftarrow (\text{mask} \,\&\, X_0) \oplus X_1$
$X_0 \leftarrow X_0 \oplus X_1$
$Y_0 \leftarrow Y_0 \oplus Y_1$
$Y_1 \leftarrow (\text{mask} \,\&\, Y_0) \oplus Y_1$
$Y_0 \leftarrow Y_0 \oplus Y_1$

**Algorithm 11** PointFromEAC(EAC $c$)

**Require:** $P(X, Y, 1)$ and $Q = \phi(P) = (X', Y', 1)$
**Ensure:** Update $Q$ with the point computed from $(P, \phi(P))$ and the Euclidean addition chain $c$.
1: **for** $i = 1 \dots \text{length}(c)$ **do**
2: SafePerm($P$, $Q$, $c_i$)
3: ZADDU($P$, $Q$)
4: **end for**
5: ZADDU($P$, $Q$)
6: **return** $Q$

## Appendix C Anatomy of a modular multiplication

Fig. 3: Anatomy of a C function computing a modular multiplication over 256-bit integers using Gnu MP (obtained from gprof).

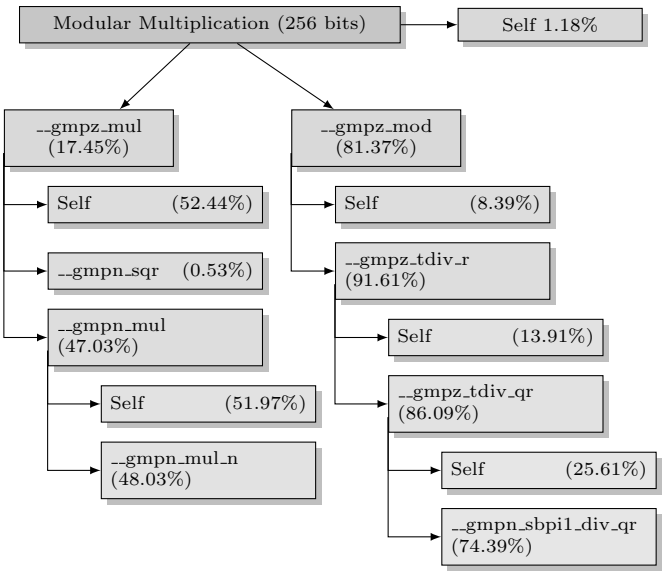


Fig. 4: Anatomy of a Java method computing a modular multiplication over 256-bit integers using Big Integer Java library on an x64 platform (obtained from Netbeans profiler).