Revista de Sistemas de Informação da FSMA
n. 13 (2014) pp. 57-67

SISTEMAS DE INFORMAÇÃO
http://www.fsma.edu.br/si/sistemas.html

TRILHA ESTUDANTIL

Performance analysis of parallel modular multiplication algorithms for ECC in mobile devices

Tiago Vanderlei de Arruda, *Mestrando em Ciência da Computação, UFSCar Sorocaba*,
Yeda Regina Venturini, *Doutor em Engenharia, UFSCar Sorocaba*,
Tiemi Christine Sakata, *Doutor em Ciência da Computação, UFSCar Sorocaba*

***Abstract*—Modular multiplication is the main operation in Elliptic Curve Cryptography (ECC) on prime finite fields. It is executed most of the time and is costly because it requires a modular reduction and a multi-precision method. Therefore, its cost increases in proportion to the ECC's key size. It could be considered as a problem for mobile devices because, despite they have more resources nowadays, their performance is still limited. Moreover, processor devices' trends are multi-core, even in the case of mobile devices. All of these facts motivated us to investigate the performance of the parallel modular multiplication on a software level on a current mobile platform. Recent studies have shown that parallel modular multiplication is effective only with big key size, like ones required for RSA, but none of them have focused on mobile devices platform. In this article, we show that, even for a smaller key size, as required for ECC, the use of parallel modular multiplication guarantees a better performance on mobile devices.**

***Index Terms*—cryptography, ECC, mobile devices, parallel modular multiplication.**

I. Introduction

USABILITY and connectivity are among the words most valued for mobile device users. Some studies find that teen users are connected nearly all waking hours of the day and this is also true for other workers. The portability of mobile devices associated with growing performance and high connectivity through wireless networks have made many applications available anytime and anywhere. Nowadays, on-line services such as e-mail, social networks, e-commerce or on-line banking are accessible using smart phones, tablets or even embedded devices, not to mention applications for sharing personal or sensors networks. All these services must provide some level of security.

Any service that needs a user (or device) identification should provide secure authentication and sometimes be followed by key exchange used for confidentiality of communication. Usually, secure authentication and key exchange are based on asymmetric cryptography algorithms, also called public key algorithms. These algorithms are based on a key pair, consisting of one secret/private and another public. The most common algorithm is the RSA [1], whose security is based on the difficulty of solving the big integer factorization problem. The security of a cryptography algorithm also depends on the private key security, which can be as great as the key size. The problem is that as the computational capacity grows, there is a need to increase the key size to ensure the same security level which, in its turn, also requires greater computational cost.

In 1985, Koblitz [2] and Miller [3] independently proposed the Elliptic Curve Cryptography algorithm (ECC). ECC is a public key algorithm whose security is based on the difficulty to compute the inverse of a big integer (scalar) multiplication of points on an elliptic curve [4].

ECC spends less computational resources than RSA, requiring a smaller key size to reach the same security level [5]. ECC is newer and more complex to understand than RSA.

The key size needed to an effective use of RSA has increased over the past years, leading secure applications to achieve higher processing loads, especially on mobile devices. Due to the strength of RSA be low for small key sizes, ECC is more suitable for devices with low computational resources or high processing loads, when performance and security are critical.

A version of ElGamal cryptography system was proposed for Elliptic Curves [6]. This system maps a confidential message on an elliptic curve (EC) as a point, and applies encryption operations on the mapped message, which results in another point corresponding to an encrypted message. The encryption operation is performed with the scalar multiplication (kP), which is defined as successive point addition operations ($P+P+...$). The point addition operation is computed by arithmetic expressions on the point coordinates, resulting in another point on the same curve (E).

The variables and coefficients of a cryptographic elliptic curve are bounded to a finite field (GF), which results in the definition of a finite abelian group [6]. Therefore, all the arithmetic operations performed must be over the chosen finite field.

When computing point addition, the field operations performed depend of the chosen coordinate system. There are several coordinate systems, which can be used to speed up the computation. These coordinate systems change the point representation and the arithmetic operations needed to compute the point addition.

The group operations used by ECC (scalar multiplication, point addition) perform a lot of operations on GF. Usually, the prime $GF(p)$ or binary $GF(2^m)$ field is applied.

This paper aims to analyze the performance of ECC defined on prime $GF(p)$ field. $GF(p)$ is defined as a set of integers less than a prime p and modular arithmetic operations on p, where the most costly required operation is the modular multiplication.

The new generations of processors are multi-core which means that an efficient implementation of ECC could be achieved through parallel computing. Some recent works [7]–[10] signal that the speeding up of parallel algorithms for the main ECC operation (modular multiplication) could only be achieved for very large operands (large p), such as used by RSA. However, it is also known that the performance of parallel algorithms is highly dependent on the processor architecture.

This work analyzes the performance of recently proposed parallel modular multiplication algorithms [8]–[10] and a parallel version of the traditional reduction algorithm proposed by Montgomery [7].

To the best of our knowledge, it is the first study on software focusing on parallel multiplication algorithms in ECC (ECC's standard key size) for a current mobile platform.

This paper is organized as follows. Section II describes the background required to understand the elliptic curve cryptography system. Section III describes the importance of modular multiplication and related works. Section IV describes the analysis of our results. Finally, Section V concludes the document describing the expected benefits of this study.

II. Elliptic curve cryptography

Elliptic curves are defined by cubic equations, similarly to the curves used for computing the circumference of an ellipse. Elliptic Curve Cryptography (ECC) is a public key encryption technique, which can be used for digital signature, encryption/decryption, and key exchange purposes.

Cryptographic applications based on ECC use an elliptic curve E defined on prime $E(GF(p))$ field or binary $E(GF(2^m))$ field (both finite field). In this text, we explore elliptic curves on $GF(p)$ since the arithmetic operations on $GF(p)$ (p prime) are more suitable for software applications [11]. The elliptic curves $E(GF(p))$ have their variables and coefficients between $[0, p-1]$, and the operation is modulo p.

A. Elliptic Curve Cryptography System

The ElGamal cryptography system [12] is based on the Diffie-Hellman key exchange algorithm [13] and can be defined on any cyclic abelian group. In elliptic curve cryptography the algebraic objects defined by (E; +) form a finite abelian group.

An ElGamal based version for Elliptic Curves was described by Hankerson et al. [6]. In their system (Figure 1), two entities, Alice and Bob, want to communicate in a secure manner over an insecure channel. First of all, Alice randomly chooses her private key $p_r \in [1, p-1]$ and generates a public key p_u as the scalar multiplication between p_r and the base point $G \in E(GF(p))$. Alice sends p_u through an insecure channel. Bob, who wants to send a message m (an ordinary text or a symmetric key) to Alice, randomly chooses a private key $r \in [1, p-1]$, maps the message in points of the chosen elliptic curve E as M, encrypt the message and send his public key (C_1) and the encrypted message (C_2) through the channel. Only the entity who owns the private key of Alice (p_r) is able to decrypt the content. The base point (or generator) G can be randomly chosen and must (as the prime p and the curve E) be known in advance by both entities.

The security of ECC algorithm is based on the difficulty to get p_r knowing $p_u = p_r G$ and G.

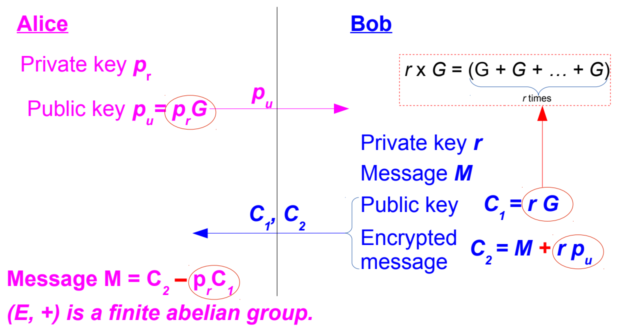
Alice
Private key p_r
Public key $p_u = p_r G$
p_u
Bob
$r \times G = \underbrace{(G + G + \ldots + G)}_{r \text{ times}}$
Private key r
Message M
Public key $C_1 = r G$
Encrypted message $C_2 = M + r p_u$
C_1, C_2
Message $M = C_2 - p_r C_1$
(E, +) is a finite abelian group.

Fig. 1. ECC based upon the ElGamal cryptosystem

It's important to note that, this system performs the point operations $P + Q$, $P - Q$ and kP, where k is a big integer and P, Q are points on an elliptic curve E. For computing $P - Q$, it's necessary to add the point P to the inverse of the point Q ($P + (-Q)$). For $P + Q$, it's necessary to add the points P and Q (Section II-B2). The scalar multiplication is represented by the product of the scalar k and the point P (kP). This operation is defined as successive point additions and doubling (Section II-B).

Figure 2 shows the hierarchy of operations performed in ECC cryptosystems. The main operation (or at least the most expensive) is the scalar multiplication, which computes several point additions/doubling according to the scalar k size (see Section II-B1). The point addition and doubling are computed according to the coordinate system on which the point is represented (details in Section II-B2). The operation on a coordinate system requires a set of arithmetic operations (e.g. addition, subtraction, multiplication, division), which are defined according to a finite field on an elliptic curve (Section II-B3). The modular arithmetic operations are applied when the chosen elliptic curve is defined on $GF(p)$.

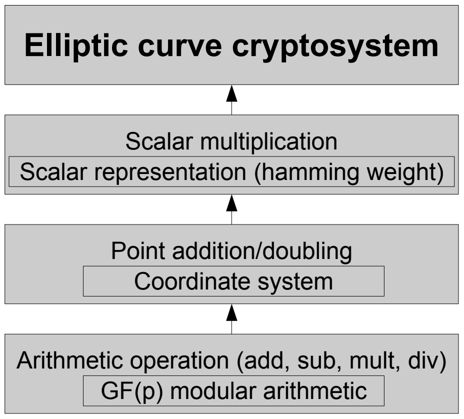
Elliptic curve cryptosystem
Scalar multiplication
Scalar representation (hamming weight)
Point addition/doubling
Coordinate system
Arithmetic operation (add, sub, mult, div)
GF(p) modular arithmetic

Fig. 2. Hierarchy of operations in ECC: ECC performs the scalar multiplication with sucessive point additions/doublings, which are performed by many modular arithmetic operations

B. Scalar multiplication

Gura et.al showed that scalar multiplication accounts for about 80% of the key calculation time in ECC [14]. The scalar multiplication is the operation of multiplying a positive integer k (usually the ECC private key) with an elliptic curve point $P \in E(GF(p))$. However, there is no way to compute this multiplication directly. So, it is made up of successive point additions [1] :

$$k \times P = \underbrace{(P + P + ... + P)}_{\text{k times}}$$

There are several proposals to compute scalar multiplication efficiently [15]. A classical and didactic method is Double-and-Add (a.k.a. Binary) [16]. This method computes scalar multiplication by point addition and doubling, which reduces the number of operations and the total cost. It scans each bit (from the least or most significant) of the scalar k. The Algorithm 1 shows the left-to-right Double-and-Add point operations. Note that the point at infinity (O) is the identity element of an elliptic curve, so that $P + O = O + P = P$ and $2O = O$.

For example, let kP be a scalar multiplication, where $k = 27$ (binary 11011) and P is a point on the elliptic curve E. The steps performed by the left-to-right Double-and-Add method for computing $27P$ are:

$$2 \times (2 \times (2 \times (2 \times ((2 \times O) + P) + P)) + P) + P$$

Note that the first doubling in the Algorithm 1 (line 4) is $Q \leftarrow 2O = O$. In practice, if the attribution in line 2 were changed to: $Q \leftarrow P$ when $k_i = 1$, or $Q \leftarrow O$ otherwise, the loop $for()$ could start with $i = n - 2$. In this case, this method performs $(n - 1)$ doubling and the number of additions depends on the number of non-zero bits of the scalar k, which is known as Hamming weight.

[1]this operations is equivalent to the exponentiation for the RSA cryptography algorithm.

Algorithm 1 Double-and-Add left-to-right scalar multiplication

```
1: procedure BINARY($k$, $P$)          ▷ binary representation of $k$ and point $P$
2:     $Q \leftarrow O$                  ▷ $O$ is the point at infinity
3:     for $i = n - 1$ to 0 do           ▷ $n$ is the bit-length of $k$
4:         $Q \leftarrow 2Q$             ▷ $2Q$ : point doubling
5:         if $k_i = 1$ then
6:             $Q \leftarrow Q + P$      ▷ $Q + P$ : point addition
7:         end if
8:     end for
9:     return $Q$
10: end procedure
```

The Hamming weight of the scalar k in the binary form is approximately $(n - 1)/2$.

Many algorithms were proposed in the literature to improve the efficiency of scalar multiplication [15]. To achieve this, it is necessary to operate on algorithms in different levels of the ECC's operation hierarchy (Figure 2):

- scalar multiplication level – identify new scalar representation or reduce hamming weigh (Section II-B1)
- point addition and point doubling operations – find out efficient coordinate system (Section II-B2)
- arithmetic operations level – speed up the performance of elliptic curve arithmetic operations (Section II-B3)

1) Scalar Representation and Hamming Weigh: The reduction of Hamming weight consequently reduces the number of point additions performed in the scalar multiplication, and thus, a performance improvement can be achieved. Some available algorithms for reducing the Hamming weight of the scalar k are [15]: *NAF*, *w-NAF*, *MOF*, *w-MOF*, *DBNS*, *MBNS*, *JSF*. The window-based variants: *w-NAF* and *w-MOF* use a pre-computed table of points during the recoding. Table I presents the approximated number of point additions and doubling of scalar multiplication for some of the cited above representations.

TABELA I
APPROXIMATED NUMBER OF POINT ADDITIONS AND DOUBLING OF SCALAR MULTIPLICATION, FOR DIFFERENT SCALAR REPRESENTATIONS

Representation	Doubling	Additions
Binary	$n - 1$	$(n - 1)/2$
NAF	n	$n/3$
w-NAF	$n + 1$	$n/(w + 1)$
MOF	n	$n/3$
w-MOF	$n + 1$	$n/(w + 1)$

2) Coordinate System: The number of arithmetic operations (addition, subtraction, multiplication, square, division, inversion) performed to compute each point addition/doubling operation varies according to the chosen coordinate system.

Figure 3 shows the geometric representation of addition and doubling point operations on an elliptic curve on the real numbers system ($\mathbb{R}$). So, let P and Q be two points on the curve E and let $-R_A$ be the point where a line through

P and Q intersects the curve. The addition $R_A = P + Q$ is the inverse of $-R_A$ over x-axis, that is, the negative of the intersection point. When $P = Q$, the doubling $R_D = P + Q = 2P = 2Q$ takes the tangent of P and finds the intersection on the elliptic curve E at a point $(-R_D)$ with the tangent line. The doubling (R_D) is the inverse of $-R_D$ about x-axis.

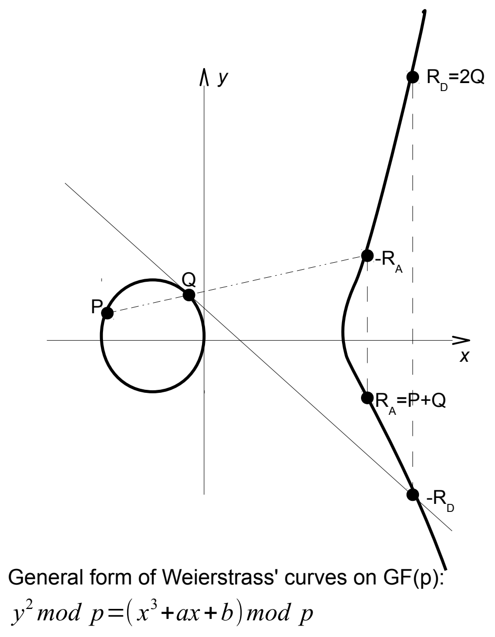
y
x
P
Q
R_D=2Q
-R_A
R_A=P+Q
-R_D
General form of Weierstrass' curves on GF(p):
y^2 mod p=(x^3+ax+b) mod p

Fig. 3. Geometric representation of point addition and doubling operations on an elliptic curve

The point addition and doubling operations using affine coordinate system are calculated according to Equation 1 and Equation 2, respectively. The variables (x, y) and coefficients a, b of the elliptic curve are defined on a finite field. Then all arithmetic operation to compute the point addition/doubling are under GF operation. For $GF(p)$ the resulting point (x_3, y_3) are computed using modular arithmetic.

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \text{ e } y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \tag{1}$$

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \text{ e } y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1 \tag{2}$$

Note that the affine coordinate system requires an inversion either in addition or in doubling operation. The inversion operation is one to two orders of magnitude slower than multiplication. However, points on a curve can be represented in different coordinate systems which do not require an inversion operation to add two points. For example, the projective coordinate systems doesn't require any inversion. However, it takes extra memory for storing temporary values and extra modular multiplications must be performed when computing the point addition and doubling operations.

Several coordinate systems have been proposed to speed up the performance of point operations. Some examples of coordinate systems defined on $GF(p)$, $p > 3$ prime proposed in the literature are: *Affine* coordinate $(\mathcal{A})$, projective coordinates: *Standard* $(\mathcal{P})$, *Jacobian* $(\mathcal{J})$, *Chudnovsky* $(\mathcal{J}^c)$, modified *Jacobian* $(\mathcal{J}^m)$ and *Mixed* [6],[17].

The number of multiplications (M), squares (S) and inversions (I) required to compute point addition and doubling operations on the cited above coordinate systems under $E(GF(p))$ curves is described by Cohen et al. [17], as shown in Table II.

TABELA II
OPERATIONS REQUIRED FOR POINT ADDITION AND DOUBLING ON $GF(p)$. M = MULTIPLICATION, S = SQUARE AND I = INVERSION. ADAPTED FROM [17].

Doubling		**Addition**	
Operation	**Cost**	**Operation**	**Cost**
$2\mathcal{P}$	$7M + 5S$	$\mathcal{P} + \mathcal{P}$	$12M + 2S$
$2\mathcal{J}^c$	$5M + 6S$	$\mathcal{J}^c + \mathcal{J}^c$	$11M + 3S$
$2\mathcal{J}$	$4M + 6S$	$\mathcal{J} + \mathcal{J}$	$12M + 4S$
$2\mathcal{J}^m$	$4M + 4S$	$\mathcal{J}^m + \mathcal{J}^m$	$13M + 6S$
$2\mathcal{A}$	$I + 2M + 2S$	$\mathcal{A} + \mathcal{A}$	$I + 2M + S$

It is possible to estimate the total number of multiplication, square, and inversion operations executed by Double-and-Add scalar multiplication (Algorithm 1) for a given size of the scalar k. Let DM, DS, DI be the number of multiplications, squares and inversions to calculate point doubling and AM, AS, AI be the number of multiplications, squares and inversions to calculate point addition in a coordinate system. Once the number of point doubling and addition operations of Algorithm 1 is approximately $(n - 1)$ and $(n - 1)/2$, where n is the size of the scalar k, the cost estimation may be computed as:

- Multiply: $(n - 1)DM + \lceil[(n - 1)/2]AM\rceil$
- Square: $(n - 1)DS + \lceil[(n - 1)/2]AS\rceil$
- Inversion: $(n - 1)DI + \lceil[(n - 1)/2]AI\rceil$

Since n is the bit-length of the scalar k, the approximated total cost of the coordinate systems can be calculated as: Projective $\mathcal{P}$: $(n - 1)(13M + 6S)$, Chudnovsky Jacobian $\mathcal{J}^c$: $(n - 1)(10.5M + 7.5S)$, Jacobian $\mathcal{J}$: $(n - 1)(10M + 8S)$, Modified Jacobian $\mathcal{J}^m$: $(n-1)(10.5M + 7S)$ and Affine $\mathcal{A}$: $(n - 1)(1.5I + 3M + 2.5S)$.

Table III shows the approximated total cost for each operation of the coordinate systems shown in the Table II (multiplications and squares) of Double-and-Add scalar multiplication with the scalar k in its binary form. Usually, the scalar considered in ECC's multiplication is the private key (e.g. $[p_r, r$ in Figure 1]). Then, the key size defines the number (n) of operation. It is also considered ECC standards to define key size (192, 224, 256, 384, 521 bits) [18].

Hankerson et al. [19] adopted projective coordinate system, because the cost for modular inversion in their underlying system was 10 times greater than the cost

TABELA III
Approximated number of multiplications and squares, for different coordinate systems.

System	Ope.	192	224	256	384	521
$\mathcal{P}$	Multiply	2483	2899	3315	4979	6760
	Square	1146	1338	1530	2298	3120
$\mathcal{J}^c$	Multiply	2006	2342	2678	4022	5460
	Square	1433	1673	1913	2873	3900
$\mathcal{J}$	Multiply	1910	2230	2550	3830	5200
	Square	1528	1784	2040	3064	4160
$\mathcal{J}^m$	Multiply	2006	2342	2678	4022	5460
	Square	1337	1561	1785	2681	3640
$\mathcal{A}$	Inversion	287	335	383	575	780
	Multiply	573	669	765	1149	1560
	Square	478	558	638	957.5	1300

of the modular multiplication. Furthermore, it's possible to note that for all the coordinate systems, the most predominant operation is the modular multiplication, so that its optimization can significantly improve the time spent on the overall cryptosystem.

3) Arithmetic Operations: An Elliptic Curve (EC) for cryptography is defined on a finite field, a Galois Field (GF). The selected GF defines a set of operations to compute the point addition/doubling. In the case of a finite prime field, $GF(p)$, these operations are modular arithmetic operations. The performance of the modular arithmetic is essential to the efficiency of the ECC algorithm on $GF(p)$.

A finite field of order p, $GF(p)$, with p prime, is defined as the set $\mathbb{Z}_p$ of integers $\{0, 1, ..., p-1\}$ and the modulo p arithmetic operations. Modular operations are always reduced to modulo p, where p is large.

The obvious way to obtain C mod M consists in dividing C by M and computing the remainder. That is, let $C, M \in \mathbb{Z}$ such that $M < C$, a modular reduction algorithm computes $R = C \bmod M$, i.e. the remainder R of the division of C by M [8], such that:

$$R = C - \left\lfloor \frac{C}{M} \right\rfloor M \tag{3}$$

The division operation needed to compute the remainder R is a costly operation, and should be avoided whenever it is possible. As an example, Table III shows that Jacobian coordinate system performs approximately 9360 operations of multiplication and square for operands with 512 bits (multi-precision integer), which would represent at least 9360 divisions for reduction, one for each modular operation.

As shown above, it is necessary to compute many modulo p arithmetic multiplication/square operations (Table II), over multi-precision integer ($i < p$, p large), to compute each point addition/doubling operation (Equation 1 and 2). The number of multiplication/square operations over big integer is defined by coordinate system, while the number of point addition/doubling operations is defined by the key size ($k < p$) and its representation (Hamming weight) (Algorithm 1), resulting into the total modular operations in the Table III. Therefore, an efficient implementation of modular multiplication is very relevant for increasing the ECC performance.

Several algorithms were proposed to compute the modular arithmetic under $GF(p)$. The next section presents some of them.

III. Modular Multiplication

When it is necessary to multiply two numbers, e.g. 128 bits each, as usual, the multiplication is done first (there are many techniques for this) to obtain a 256-bit (”double precision”) number. Then, as we consider $GF(p)$, it is necessary to apply the modular reduction for a given prime p. In many situations, the ”schoolbook” technique everyone learns in school is suitable, but sometimes what is required is a more efficient execution, such as the one where many reductions are done by the same modulo.

The most well known techniques are the Montgomery and Barret reduction algorithms. The Montgomery reduction algorithm [20], [21] is extensively used in public key cryptography to compute modular operation. The algorithm modifies the integer representation to Montgomery's residue form with radix r. The chosen radix is usually a power of 2, such that the reduction modulo r is performed more efficiently using native shift instructions of the underlying hardware [8], [22]. This algorithm is more suitable when several modular operations with the same modulo are performed, because of the cost of preprocessing operations [22].

The Barret reduction algorithm [23], [24] also performs the division modulo a number power of 2, instead of modulo p. This algorithm pre-computes an approximation of the inverse of M to approximate the quotient Q, as shown in the Equation 4.

$$Q = \left\lfloor \left\lfloor \frac{C}{r^n} \right\rfloor v / r^n \right\rfloor \tag{4}$$

The pre-computed approximation is: $v = \lfloor r^{2n}/M \rfloor$ which makes possible the efficient computation of the approximation Q of $\lfloor C/M \rfloor$. The fully reduced remainder can be achieved by computing $C - QM$, followed by some simple operations [8].

As it can be seen in Table II, the projective coordinate systems avoid the inversion, on the other hand, they perform more modular multiplications and squares than the Affine coordinate system. Nevertheless, Table III shows the theoretical approximated number of operations in some available coordinate systems, demonstrating how the optimization of modular multiplication on these systems is essential. Montgomery and Barret proposed an efficient algorithm to compute the reduction modulo p for big integer (result of multiplication). Therefore, the optimization of such an operation is viable. Other operations are faster than the multiplication, thus their parallelism isn't a priority [25].

Most proposals in the literature have explored the parallelism inherent to the integer multiplication (in its binary form) on modular multiplication, because it enables independent partial computations to be performed in parallel,

TABELA IV
Brief description of some parallel modular multiplication algorithms evaluated on MIMD architectures

Algorithm	**Description**
Parallel Montgomery	Parallelizes the sub-operations of sequential Montgomery.
Bipartite	Partitions one operand. Run in parallel on two cores.
Tripartite	Partitions two operands. Increases the overhead when more parallel processes are launched.
k-ary Multipartite v1	Partitions two operands. Without increase in the synchronization costs. Constant overhead.
k-ary Multipartite v2	Partitions two operands. Variation of k-ary Multipartite v1.

which at the end are added together. The Montgomery reduction algorithm is commonly adopted in these implementations because it enables the modular reduction to be efficiently performed on modular arithmetic.

Chen et al. [26] proposed a parallel extension of a variant (SOS) of Montgomery modular multiplication algorithm in software, for a prototype of processor architecture with distributed local memory and message-passing communication. They proposed a task balancing model in which the multiplication is performed by parallel processes and the operations are computed in task blocks, each associated to a processor in a circular manner. Experiments were performed with 2, 4 and 8 processing cores.

The parallelism inherent to the integer multiplication was explored by Baktir et al. [7]. In their work, it was created a parallel function to perform the multiplication, which was used to compute the product operations inside the Montgomery modular multiplication. The partial product accumulation was achieved in a binary tree fashion. Experiments were performed in general purpose processors with 1, 2, 4, 6 cores, for 1024, 2048, 4096, 8192, 16384, 32768 bit operands and compared with the single-core implementation of the proposed algorithm and a variant of (CIOS) Montgomery multiplication. In their work, the authors obtained speedups of up to 81% for bit-lengths of 4096 in two cores. There were performance improvements for the implementations on 4 and 6 cores architectures.

The Bipartite modular multiplication algorithm was proposed by Kaihara et al. [9] for computing the modular multiplication from left-to-right and right-to-left, using Montgomery and Barret modular multiplications. The modular multiplication $R = AB \bmod M$ is computed using a new modulo M residue class representation. The Bipartite algorithm divides the multiplier B in two blocks, which are processed separated in parallel. The Montgomery modular multiplication algorithm scans the multiplier from the least significant bits while the Barret scans from the most significant [9].

The Tripartite modular multiplication algorithm [10] maximizes the parallelism level achieved relatively to Bipartite. This algorithm uses the Karatsuba's method for integer multiplication, and enables the parallel computation of three terms. This algorithm has a high synchronism overhead and a sub-quadratic complexity.

Recently, Giorgi et al. [8] proposed two versions of a k-ary Multipartite modular multiplication as a generalization of Bipartite and Tripartite methods, where k is the number of partitions in which the operands are divided. In their work, two versions were proposed. The k-ary Multipartite algorithms doesn't have additional synchronism overhead as the Tripartite modular multiplication, and another optimizations were applied. In k-ary Multipartite v1, the authors optimized the computation of the Q-value (see [8] for more information), so that all the partial products with same weight were added before a single call to PMR (Partial Montgomery Reduction) or PBR (Partial Barret Reduction) to compute the remainder. When no reduction is needed, the algorithm includes the product into the remainder list. And finally, after sum all the partial remainders, some final subtractions are performed to fully reduce the result. The k-ary Multipartite v2 algorithm doesn't call PMR and PBR for each weight, instead, it computes only the Q-value and includes in a Q-value list. These Q-values are added and finally, the remainder is achieved by subtracting the product of the final Q-value by P from the sum of the partial products. The algorithm was compared with Montgomery, Barret, Bipartite and Tripartite multiplication, running on 1, 3, 4, 6, 8 parallel threads and operands with sizes between 1024 and 16384 bits. The test was carried out on two 64 bits processors with four cores each. The experiments showed the bipartite algorithm as the fastest algorithm for greater keys, because of its low parallel complexity. Bipartite can be suitable for more than 2 cores, since the key size is increased. Their k-ary multipartite algorithms seemed a good alternative for implementations with operands smaller than 2^{13} bits.

Table IV shows a brief description of the above mentioned algorithms, which were evaluated on MIMD multicore architectures. Giorgi et al. [8] implemented a library accomplishing all these algorithms, and gently provided it. Benchmarks were run on theese two architectures, and the results are evaluated in this paper.

The related parallel modular multiplication algorithms weren't proposed for a specific coordinate system because they optimize the modular multiplication which is one of the underlying operations to the coordinate system doubling/addition point operations.

As the library used by Giorgi is built over GMP (GNU Multi-precision library), which makes use of optimized instructions for several architectures, this work evaluates if it is viable to parallelize the algorithm for keys actually used (or will be used in a near future) for elliptic curve cryptography purposes when executed by a mobile device in a 32 bits quad-core processor.

TABELA V
Timings in μs for some parallel modular algorithms on 1, 2, 3, 4 threads, and operands ranging from 128 to 4096 bits on an Intel Core I7 architecture. The best timings per key are bolded, and the best timings per Thread are in gray background.

	Algorithm	**128**	**256**	**384**	**512**	**768**	**1024**	**1536**	**2048**	**3072**	**4096**
1 Thread	Best seq.	**0.86**	**0.47**	**0.61**	**0.83**	1.54	2.19	4.52	6.56	13.01	22.67
2 Threads	Montgomery	5.73	2.52	2.73	2.86	3.12	3.67	5.65	7.02	11.60	15.95
	Bipartite	2.21	0.85	0.97	1.08	**1.36**	1.83	3.48	4.63	8.44	11.83
3 Threads	Montgomery	2.53	2.61	2.74	2.65	2.86	3.10	4.13	5.24	8.18	10.50
	2-ary Multi. v1	1.78	1.93	1.88	2.01	2.04	2.51	3.44	4.27	**6.88**	**9.02**
	2-ary Multi. v2	1.01	1.17	1.10	1.13	1.39	**1.69**	2.99	4.03	7.38	10.16
4 Threads	Montgomery	2.74	2.81	2.61	2.72	3.02	3.73	4.29	5.37	7.54	9.85
	Bipartite	2.88	2.88	2.99	3.05	3.21	3.57	4.01	4.64	7.40	9.21
	4-ary Multi. v1	4.26	1.95	1.98	2.11	2.39	2.56	3.38	4.16	6.89	9.13
	4-ary Multi. v2	1.16	1.23	1.32	1.38	1.63	1.94	**2.95**	**3.78**	7.76	10.93

TABELA VI
Timings in μs for some parallel modular algorithms on 1, 2, 3, 4 threads, and operands ranging from 128 to 4096 bits, on a Freescale IMX6Quad architecture. The best timings per key are bolded, and the best timings per Thread are in gray background.

	Algorithm	**128**	**256**	**384**	**512**	**768**	**1024**	**1536**	**2048**	**3072**	**4096**
1 Thread	Best seq.	**1.22**	**1.84**	2.78	4.02	5.06	6.01	11.52	19.42	39.18	64.93
2 Threads	Montgomery	6.28	6.81	7.60	4.30	5.62	7.33	12.15	19.87	34.24	52.99
	Bipartite	2.95	3.52	3.86	2.54	3.51	4.90	8.89	14.86	26.72	42.71
3 Threads	Montgomery	5.89	3.13	3.19	3.44	4.78	5.98	8.62	12.71	20.18	31.45
	2-ary Multi. v1	4.74	5.09	2.69	3.02	3.69	5.24	7.42	11.15	18.47	29.10
	2-ary Multi. v2	3.10	3.62	**2.08**	**2.48**	**3.31**	4.80	7.75	12.55	22.04	35.45
4 Threads	Montgomery	6.09	3.22	3.39	3.77	4.16	5.84	8.28	11.93	19.13	29.62
	Bipartite	6.87	3.55	3.87	4.22	4.80	5.96	7.68	**10.37**	**16.81**	**26.18**
	4-ary Multi. v1	4.62	2.81	2.86	3.01	4.07	4.98	7.33	10.74	17.72	27.90
	4-ary Multi. v2	3.52	2.01	2.22	2.55	3.39	**4.37**	**7.24**	12.21	20.95	34.52

IV. Results

All of the parallel modular multiplication algorithms evaluated in this paper are part of the C++ library developed and provided by the authors of [8]. This library accomplishes the Parallel Montgomery, Bipartite, Tripartite, Multipartite v1 and v2 algorithms for shared memory MIMD architectures. The basic multi-precision arithmetic (e.g. addition, multiplication, subtraction, division) is performed using GMP (GNU Multi-precision Library), which is very suitable for cryptographic applications. It owns hardware optimizations for some architectures and chooses the best algorithm among Basecase, Tom-Cook and FFT, according to the operands' size. Giorgi et al. [8] used the OpenMP API to launch parallel tasks, and evaluated the algorithms of 1024 to 16384 bit-length operands on an architecture embedding two Intel Xeon X5650 Westmere processors, with six cores each, running at 2.66 GHz. The timings were collected for 1, 3, 4, 6 and 8 threads.

In this paper, experiments were performed on a Sabre Lite development board, which has a Freescale IMX6Quad processor, with 4 cores, running at 1.00 GHz. Further, experiments were also performed on PC accomplishing an Intel Core I7 processor, with 4 physical and 4 virtual cores, running at 3.40 GHz. Some parallel modular multiplication algorithms were evaluated for operands from 128 to 4096 bit-length on 1, 2, 3 and 4 threads. The creation of n threads in OpenMP doesn't imply the use of n distinct cores by the runtime system, which manages the parallel scheduling.

According to Pacheco [27], the minimum elapsed time is usually reported when timing a parallel program, once it is unlikely that some outside the event could make it run faster than its best possible runtime. In this work, the minimum time taken to run each algorithm for operands ranging from 128 to 4096 bits for over 1000 runs were reported, and will be analysed later.

Tables V and VI show the benchmark results for Intel Core I7 and Freescale IMX6Quad architectures, respectively. It can be seen the substantial difference between the parallel and sequential timings, when the size of operands increases. The best timings achieved for a certain number of threads and with different operands' size are in gray background, and the best timings achieved for a specific operand bit-length are bolded. The "Best seq." line is the best time between the serial Montgomery and GMP default modular multiplication.

On Intel Core I7 architecture, the "Best seq." algorithm achieved the best timings for operands ranging from 128 to 512 bit-length. Parallel algorithms were faster for operands larger than 512 bit-length. As the bit-length increases the best timing is achieved with more threads. However, for 3072 and 4096 bit-length the benchmark with 3 threads performed faster than with 4 threads.

It is shown that for the Freescale IMX6Quad architecture, the "Best seq." algorithm achieved the best timings for small operands with 128 and 256 bit-length. Parallel algorithms achieved the best timings for operands larger than or equal to 384 bit-length.

On 32-bit Freescale, parallel algorithms achieved better timings than the sequential with operands greater than 384 bit-length, while on 64-bit Intel Core I7, this threshold was 768-bits. In both architectures, some algorithms achieved a very poor timing for 128-bit operands.

The 2-ary Multi. v2 algorithm on the Freescale architecture achieved the best timings for 384 to 768-bit operands running with 3 threads. This range of operands' size is suitable for ECC nowadays. When larger operands is used, as required by RSA, 4 threads performed better.

Bipartite modular multiplication has a high synchronization overhead but at the same time its parallel complexity is the cheapest on 4 threads [8]. Despite its high overhead, the arithmetic complexity dominates the processing time for very large operands (see Table VI). The same behavior was reported in [8] when the number of threads was equal to the number of physical cores in a processor.

Evaluating operands with 512-bit length on 32-bit Freescale architecture, we concluded that despite the fact that the best timings were achieved on 3 threads, the best timings for 2 or 4 threads were very close. The best algorithm on 3 threads was the 2-ary Multi. v2, and on 4 threads, the best algorithms were the 4-ary Multi. v2 and Bipartite.

Parallel Montgomery modular multiplication did not achieve the best timing for any operand's size and number of threads. This algorithm has higher complexity associated to many synchronizations (parallel overhead) [8].

The parallel program can be at most p times faster than the serial program, where p is the number of cores [27]. The speedup (Equation 5) is used to measure the best it can reach:

$$S = T_{serial}/T_{parallel} \tag{5}$$

As explained by Pacheco [27], it's unlikely to get linear speedup because of the overhead related to the synchronism costs. Thread launching and memory operations also imply in additional overhead, which affect the speedup of a parallel program [8].

Another important concept is the efficiency, which is described by Pacheco [27]. This method allows us to identify how efficiently the cores can be used. If the efficiency is 1, the parallel program has linear speedup. Efficiency is given by $E = S/p$.

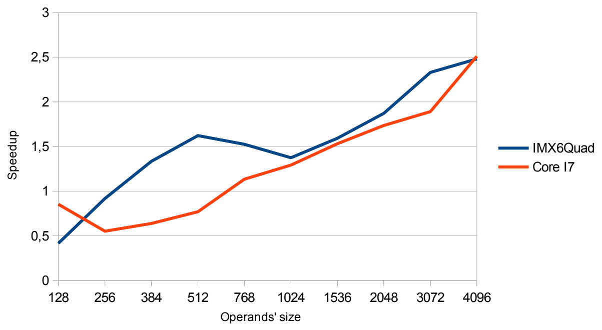

Fig. 4. Best speedups on IMX6Quad and Core I7

Figure 4 shows the best speedups achieved for operands ranging from 128 to 4096 bit-length, on Freescale IMX6Quad and Intel Core I7 architecture. The speedup of IMX6Quad increased substantially from 128 to 512 bits, fell slightly until 1024 operand's size, and increased gradually for longer operand's size. The speedup of Intel Core I7 started with a sharp decrease for 256-bit operands, increasing gradually until the 4096-bit operands. The parallel algorithms have taken the most advantage on IMX6Quad, once its speedup line exceeds 1 before the Core I7 speedup line. The Core I7 architecture evaluated works on higher frequency and has more RAM than the IMX6Quad architecture. As in this paper the goal is evaluate the improvement of parallel algorithms on mobile devices, it shows a positive trend on constrained devices, which besides having some bottlenecks, achieved in our experiments better improvement than Core I7 for parallel algorithms.

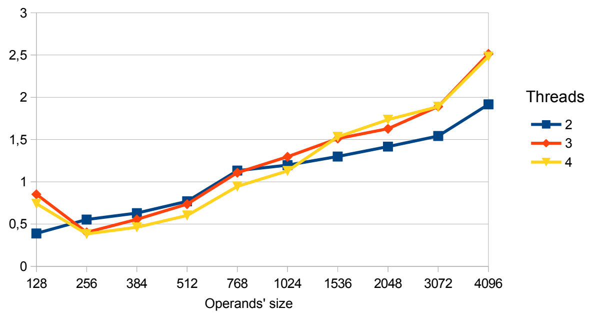

Fig. 5. Speedups vs. operands' size for 2, 3 and 4 threads on **Core I7**

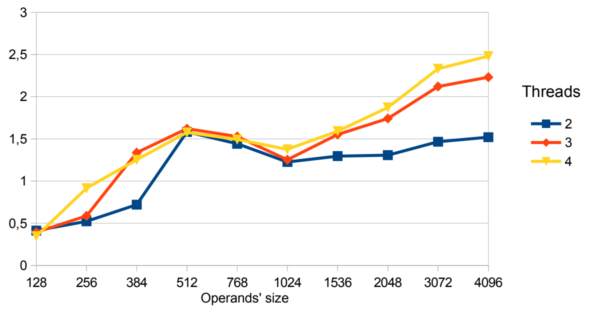

Fig. 6. Speedups vs. operands' size for 2, 3 and 4 threads on **IMX6Quad**

.

Figures 5 and 6 show the speedups achieved by the parallel algorithms for some operand sizes. The best timings by number of threads are considered to compute these speedups ($S = T_{best\ seq.}/T_{best\ parallel}$).

As shown in Figure 5 (Core I7), modular multiplication between 128 to 512-bit operands are faster when performed by a serial algorithm, once the best speedups are less than 1 for these sizes. For operands greater than 758, it's possible to see that the best speedups achieved on Core I7 architecture were reached by 3 and 4 threads implementations. But the efficiency of 3 threads fastest algorithm is better than the 4 threads.

Figure 6 (IMX6Quad) shows that 4 threads speedup was slightly lower than the 3 threads speedup for 384, 512, 768-bit operands. Nevertheless, the speedup for 4 threads was growing for operands from 1024 to 4096 bits, remaining higher than for 3 threads. As the speedup achieved on IMX6Quad architecture for 128 and 256-bit operands were less than 1, they are best computed by serial algorithms on this architecture.

V. Conclusions

NOWADAYS, most people rely on their mobile devices for their daily activities (calendar, e-mail, news, documents, tasks, entertainment and so on). For them, a high speed network connectivity together with a high performance device and a certain level of security is mandatory. In this context, ECC is a good choice for providing security for mobile devices because it requires less computation resources compared to RSA.

This paper emphasizes the importance of modular multiplication for an ECC algorithm. Therefore, a way to improve the ECC's performance is to implement some parallel version of this relevant operation.

As the projective coordinate systems are faster than the affine coordinate system, due to the replacement of modular inversion by additional modular multiplication and square, this work presented the estimated cost when using these systems, as well, the theoretical cost according to the size of the operands. It was shown that for all sizes, the number of modular multiplications is almost 2 times the number of squares.

Several algorithms for parallel modular multiplication were proposed in software, and Giorgi et al. [8] has created a C++ library including some of them. In this work, we evaluate the same algorithms as [8], using smaller keys, as used by ECC systems. Experiments were performed on a Sabre Lite development board, with a 32-bit Freescale IMX6Quad processor to evaluate the performance of these algorithms in a current mobile platform. Also, the same experiments were reproduced on a PC with a 64-bit Intel Core I7 processor to compare the results.

We show that, on the 32-bit mobile development board, operands with at least 384 bits achieve faster execution when performed by parallel algorithms, while on Core I7 the parallel timing is advantageous only for operands with at least 768 bits (twice the length in Freescale).

Analysing the speedups, it was possible to identify the behavior of the algorithms, according to the platform and the number of threads used.

On Core I7 architecture, the 3 threads best algorithms achieved a high speedup and better efficiency than 4 threads algorithms. It is possible to conclude that on this architecture, the best 3 threads algorithms are preferable when high speed and efficient usage of cores are needed, for the operands greater than 768 bits evaluated.

Parallel algorithms achieved speedups on IMX6Quad higher than on Core I7 architecture. On this platform, if time is critical, the 4 threads best algorithms should be used. Otherwise, if the efficient core usage is critical, then the 3 threads best algorithms are the best choice.

The experiments have shown the dependence of the parallel algorithm related to processor architecture, specially when the problem size (bit-length) can change. The GMP library used already includes a hardware optimization such that the best algorithm is chosen for each processor in a published list. Nevertheless, algorithms developed based on this library request new performance evaluation. This is because the parallel algorithms' performance is impacted by both algorithm complexity and parallelism's overhead.

The results show that even using smaller key size, the performance of parallel modular multiplication outperforms the sequential implementation. As the modular multiplication operation is executed a huge amount of times and it is the most costly operation in ECC, our results guarantee the improvement of the ECC system on a mobile platform.

This result confirms that it is worth using parallel implementation of modular multiplication for ECC in mobile devices.

VI. Acknowledgments

THIS work was partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq) Universal grant 482342/2011-0 and the Coordination for the Qualification of Higher Education Staff (CAPES). We would like to thank Pascal Giorgi, Laurent Imbert and Thomas Izard for providing us their library and for their relevant contribution on this area. We thank Abedin R. Talouki for his contributions to this work.

Referências

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978, ISSN: 0001-0782. DOI: 10.1145/359340.359342. [Online]. Available: http://doi.acm.org/10.1145/359340.359342.

[2] N. Koblitz, "Elliptic curve cryptosystems," English, *Mathematics of Computation*, vol. 48, no. 177, pages, 1987, ISSN: 00255718. [Online]. Available: http://www.jstor.org/stable/2007884.

[3] V. Miller, "Use of elliptic curves in cryptography," English, in *Advances in Cryptology - CRYPTO '85 Proceedings*, ser. Lecture Notes in Computer Science, H. Williams, Ed., vol. 218, Springer Berlin Heidelberg, 1986, pp. 417–426, ISBN: 978-3-540-16463-0. DOI: 10.1007/3-540-39799-X_31. [Online]. Available: http://dx.doi.org/10.1007/3-540-39799-X%5C_31.

[4] I. Kovalenko and A. Kochubinskii, "Asymmetric cryptographic algorithms," English, *Cybernetics and Systems Analysis*, vol. 39, no. 4, pp. 549–554, 2003, ISSN: 1060-0396. DOI: 10.1023/B:CASA.0000003504.91987.d9. [Online]. Available: http://dx.doi.org/10.1023/B:CASA.0000003504.91987.d9.

[5] K. Gupta and S. Silakari, "Ecc over rsa for asymmetric encryption: a review," *International Journal of Computer Science Issues*, vol. 8, no. 3, pp. 370–375, 2012.

[6] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003, ISBN: 038795273X.

[7] S. Baktir and E. Savas, "Highly-parallel montgomery multiplication for multi-core general-purpose microprocessors," in *Computer and Information Sciences III*, E. Gelenbe and R. Lent, Eds., Springer London, 2013, pp. 467–476, ISBN: 978-1-4471-4593-6. DOI: 10.1007/978-1-4471-4594-3_48. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-4594-3%5C_48.

[8] P. Giorgi, L. Imbert, and T. Izard, "Parallel modular multiplication on multi-core processors," in *IEEE Symposium on Computer Arithmetic*, A. Nannarelli, P.-M. Seidel, and P. T. P. Tang, Eds., IEEE Computer Society, 2013, pp. 135–142, ISBN: 978-1-4673-5644-2.

[9] M. Kaihara and N. Takagi, "Bipartite modular multiplication method," *IEEE Trans. Comput.*, vol. 57, no. 2, pp. 157–164, Feb. 2008, ISSN: 0018-9340. DOI: 10.1109/TC.2007.70793. [Online]. Available: http://dx.doi.org/10.1109/TC.2007.70793.

[10] K. Sakiyama, M. Knezevic, J. Fan, B. Preneel, and I. Verbauwhede, "Tripartite modular multiplication.," *Integration*, vol. 44, no. 4, pp. 259–269, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/integration/integration44.html%5C#SakiyamaKFPV11.

[11] J. Portilla, A. Otero, E. de la Torre, T. Riesgo, O. Stecklina, S. Peter, and P. Langendörfer, "Adaptable security in wireless sensor networks by using reconfigurable ecc hardware coprocessors," *International Journal of Distributed Sensor Networks*, vol. 2010, 2010.

[12] T. El Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Proceedings of CRYPTO 84 on Advances in cryptology*, Santa Barbara, California, USA: Springer-Verlag New York, Inc., 1985, pp. 10–18, ISBN: 0-387-15658-5. [Online]. Available: http://dl.acm.org/citation.cfm?id=19478.19480.

[13] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976.

[14] N. Gura, A. Patel, A. Wander, H. Eberle, and S. Shantz, "Comparing elliptic curve cryptography and RSA on 8-bit CPU's," in *Cryptographic Hardware and Embedded Systems-CHES*, 2004, pp. 925–943.

[15] E. Karthikeyan, "Survey of elliptic curve scalar mutiplication algorithms," *Int. J. Advanced Networking and Applications*, vol. 04, no. 2, pp. 1581–1590, 2012, ISSN: 0975-0290. [Online]. Available: http://www.ijana.in/papers/V4I2-8.pdf.

[16] O. Ahmadi, D. son, and F. Rodríguez-Henríquez, "Parallel formulations of scalar multiplication on koblitz curves," *j-jucs*, vol. 14, no. 3, pp. 481–504, Feb. 1, 2007.

[17] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren, *Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second Edition.* Chapman & Hall/CRC, 2006, ISBN: 1439840008, 9781439840009.

[18] S. for Efficient Cryptography. 2010, *SEC 2: Recommended Elliptic Curve Domain Parameters. (January 2010)*, Retrieved November 11, 2013 from http://www.secg.org/download/aid-784/sec2-v2.pdf.

[19] D. Hankerson, J. Hernandez, and A. Menezes, "Software implementation of elliptic curve cryptography over binary fields," English, in *Cryptographic Hardware and Embedded Systems – CHES 2000*, ser. Lecture Notes in Computer Science, Ç. Koç and C. Paar, Eds., vol. 1965, Springer Berlin Heidelberg, 2000, pp. 1–24, ISBN: 978-3-540-41455-1. DOI: 10.1007/3-540-44499-8_1. [Online]. Available: http://dx.doi.org/10.1007/3-540-44499-8%5C_1.

[20] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[21] R. Afreen and S. C. Mehrotra, "A review on elliptic curve cryptography for embedded systems," *CoRR*, vol. abs/1107.3631, 2011.

[22] Ç. K. Koç, "Montgomery reduction with even modulus," English, *IEE Proceedings - Computers and*

Digital Techniques, vol. 141, 314–316(2), 5 Sep. 1994, ISSN: 1350-2387. [Online]. Available: http://digital-library.theiet.org/content/journals/10.1049/ip-cdt%5C_19941291.

[23] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology - CRYPTO' 86*, ser. Lecture Notes in Computer Science, A. Odlyzko, Ed., vol. 263, Springer Berlin Heidelberg, 1987, pp. 311–323, ISBN: 978-3-540-18047-0. DOI: 10.1007/3-540-47721-7_24. [Online]. Available: http://dx.doi.org/10.1007/3-540-47721-7%5C_24.

[24] M. Brown, D. Hankerson, J. Lopez, and A. Menezes, "Software implementation of the nist elliptic curves over prime fields," in *TOPICS IN CRYPTOLOGY - CT-RSA 2001, volume 2020 of LNCS*, Springer, 2001, pp. 250–265.

[25] R. Laue and S. Huss, "Parallel memory architecture for elliptic curve cryptography over $\mathbb{GF}(p)$ aimed at efficient fpga implementation," English, *Journal of Signal Processing Systems*, vol. 51, no. 1, pp. 39–55, 2008, ISSN: 1939-8018. DOI: 10.1007/s11265-007-0135-9. [Online]. Available: http://dx.doi.org/10.1007/s11265-007-0135-9.

[26] Z. Chen and P. Schaumont, "A parallel implementation of montgomery multiplication on multicore systems: algorithm, analysis, and prototype," *Computers, IEEE Transactions on*, vol. 60, no. 12, pp. 1692–1703, 2011, ISSN: 0018-9340. DOI: 10.1109/TC.2010.256.

[27] P. Pacheco, *An Introduction to Parallel Programming*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, ISBN: 9780123742605.

Tiago Vanderlei de Arruda possui graduação em Análise e Desenvolvimento de Sistemas pela Faculdade de Tecnologia de Sorocaba e cursa o Mestrado em Ciência da Computação da Universidade Federal de São Carlos, campus de Sorocaba.

Yeda Regina Venturini possui Doutorado em Engenharia Elétrica pela Escola Politécnica da Universidade de São Paulo e atualmente é professora adjunto da Universidade Federal de São Carlos, campus de Sorocaba.

Tiemi Christine Sakata possui Doutorado em Ciência da Computação pela Universidade Estadual de Campinas e atualmente é professora adjunto da Universidade Federal de São Carlos, campus de Sorocaba.