Improving the Efficiency of Point Arithmetic on Elliptic Curves Using ARM Processors and NEON

Pham Van Luc[1], Hoang Dang Hai[2], and Leu Duc Tan[3]

(Corresponding author: Pham Van Luc)

Faculty of Electronic Posts and Telecommunications Institute of Technology[1]

Km10, Nguyen Trai , Ha Dong District, Hanoi, Vietnam

Email: pvluc@bcy.gov.vn

Posts and Telecommunications Institute of Technology Ha Noi, Viet Nam[2]

Leu Duc Tan Academy of Cryptography Techniques Ha Noi, Viet Nam[3]

(Received Aug. 31, 2021; Revised and Accepted Jan. 13, 2022; First Online Feb. 14, 2022)

Abstract

Point arithmetic operations, especially scalar point multiplication, are important for cryptosystems on elliptic curves. These operations have a large computational overhead, which significantly affects the efficiency and speed of the cryptosystems. Several studies proposed methods to reduce the number of operations and the computational cost. However, few studies investigated hardware characteristics to improve the efficiency of point arithmetic operations by reducing the number of intemediate calculations. In this paper, we propose combining the Karatsuba algorithm with dual multiplications, which are performed in parallel on ARM processors with the NEON component. We propose some improved algorithms for point arithmetic operations by grouping pairs of multiplications or pairs of squarings to reduce intermediate calculations. Experimental results shown an efficiency increase of from 20% to 30% for point arithmetic operations (point addition, doubling and scalar point multiplication) and from 10% to 20% for the cryptographic primitive operations in ECDH and ECDSA protocols on ARMv7 and ARMv8 embedded microprocessors.

Keywords: ARM Processor; Elliptic Curve Cryptography; NEON Component; Point Addition; Point Doubling; Scalar Point Multiplication

1 Introduction

Arithmetic operations in the finite fields, especially scalar multiplication, play an important role in the elliptic curve cryptography (ECC) systems. Several standards have emerged, such as digital signature standard (DSS) [22], IEEE 1363 [27], and NIST [23], which recommend using finite fields for the digital signature algorithm on the elliptic curves. The most important ECC operation is the scalar point multiplication kP, where k is an integer, P is a point on the elliptic curve. To perform the multiplication, we can use the point addition and the point doubling multiplication. Point multiplications are more complex, thus, they take up the most computation time. The efficiency of the ECC cryptosystems mainly depends on the complexity and the speed of these operations [7, 14, 29]. The solution for these two opposing objectives is challenging because the efficiency depends further on the computing power of the deployment platform, especially for constrained hardware platforms with low-performance processors.

For improving the efficiency of the arithmetic operations on elliptic curves, theoretical studies often focused on: 1) reducing the number of the point addition and point doubling operations required in scalar multiplications [14]; 2) increasing the efficiency of the point addition and doubling formulas by exploiting the methods of point representation on elliptic curves [13]; 3) reducing the computational cost by improving the efficiency of the arithmetic operations [11, 20]. In fact, these methods can be combined together for obtaining a better performance on some hardware platform, such as in [4, 9, 16]. The Karatsuba (KA) [13] algorithm was the first to efficiently perform the multiplication of integers with a low computational complexity. However, for deploying on hardware platforms with limited processing capacity or on new hardware platforms, the algorithm needs to be improved for taking the full advantage as shown in some studies, such as [12, 16, 20, 24, 28].

In recent years, there is an increasing number of embedded microprocessors that provide single instruction multiple data (SIMD) capabilities to support parallel processing on dedicated modules. The commonly used ARM processor family is the Cortex-A architecture [1, 9, 17]. Most

Cortex-A architectures include the NEON component, which provides SIMD vector instructions. The NEON component allows parallel processing of SIMD instructions, resulting in an increased computation speed. A number of studies attempted to exploit the NEON instructions on ARM processors for increasing the computation speed of the implemented cryptographic algorithms, such as in [3, 5, 9, 26]. Some recent studies, such as in [11, 12, 15, 20], shown certain results by applying the SIMD architecture with NEON for implementing arithmetic operations on ECC. The NEON instructions, in particular the decomposition of instructions for performing parallel tasks in the SIMD architecture, enabled considerable advantages. However, several remaining issues are the cost of intermediate products, the challenge of handling propagation carries in prime fields, the problem of redundant calculations, etc. A large number of redundant intermediate calculations can cause a considerable degradation of the computation performance.

According to our survey, we see that the problem of reducing redundant intermediate calculations has not been fully investigated. There is a need for exploiting the specific features of the new hardware platform with NEON, such as ARM Cortex-A's, to reduce redundant instructions. This possibility can help to increase the computation speed, and therewith improves the efficiency of the arithmetic operations on elliptic curves. In particular, the combination of the well-known Karatsuba algorithm with ARM's advanced hardware platform can lead to a higher performance, but has not received much attention. The use of much read and write instructions between the memory and the NEON component can cause a considerable computational overhead, which can result in the performance degradation of the ECC cryptosystems.

This paper proposes a method to improve the efficiency of the point arithmetic operations (including the point addition, the point doubling, and scalar point multiplication) on elliptic curves. Our method focuses on gaining the SIMD hardware features of ARM processors with the integrated NEON component to speed up the point arithmetic operations. The key features of the method are: 1) combining the ordinary multiplication (the operand-scanning method) with the Karatsuba's multiplication algorithm for long operands; 2) implementing the parallel multiplications (the dual multiplications) on the prime fields in the combination with pairing to reduce the overhead of reading and/or writing data between the internal memory and the NEON component; 3) using an available large number library RELIC to speed up the computation. The proposed method has been fully integrated into the calculations of ECDH (Elliptic Curve Diffie Hellman) and ECDSA (Elliptic Curve Digital Signature Algorithm) protocols on GF(p) fields with sizes of 256 bits, 384 bits, and 521 bits.

The rest of the paper is structured as follows. Section 2 presents related studies. Section 3 gives a brief overview of elliptic curves over finite fields. Section 4 presents our methods to improve the efficiency of the point arithmetic. Experimental results are given in Section 5. Finally, Section 6 is the conclusion.

2 Related Work

In this section, we present some typical related studies, which apply the SIMD architecture and the NEON component in order to increase the speed of the arithmetic operations such as multiplications, squarings, and modulo operations in the finite fields on elliptic curves.

The authors in [3] suggested using NEON in the Cortex-A8 processor to speed up the computation of the shared secret key. A simplified radix representation method was used to perform NEON-based multiplications to speed up arithmetic operations on Curve25519 and Ed25519 curves. The NEON vector was used for two independent multiplications: a point multiplication as well as a single multiplication. The paper in [9] presented an algorithm to improve the efficiency of calculating the scalar multiplications on ECC with the focus on side-channel protection. Scalar multiplications were based on the proposed GLV (Gallant-Lambert-Vanstone) method, and used interleaved ARM-NEON instructions to perform 128-bit independent multiplications in parallel. The study in [26] implemented an attribute-based encryption scheme using Cortex-A9 and NEON. The authors proposed a method to exploit the ability to compute bilinear pairings on the ellictic curves. The authors in [5] presented a polynomial multiplier using NEON in the ARM Cortex-A8, A9, and A15 processors. The NEON vector was used to speed up the computation in the binary fields on the elliptic curves. The polynomial multiplier used two 8-bit vectors to form 128-bit products. These basic multipliers can be used in point multiplications. Some studies focused on the topic of implementing cryptographic primitive functions on the SIMD structure by using the NEON vector to improve the computational efficiency on ECC, such as [4,17]. In [4], a parallel version of the Montgomery interleaved multiplication algorithm was proposed using the extended vector instructions of SIMD and NEON. The Montgomery multiplication was splitted into two parallel operations. In [17], the authors proposed to use the interleaved ARM and NEON instructions to speed up the multiplications on $\mathbb{F}_{p^2}$, thereby they could speed up four-dimensional scalar multiplications on the Four$\mathbb{Q}$ twisted Edwards curves [17].

As we can be, using NEON to perform parallel operations can provide good efficiency. However, some typical problems are still remaining as follows. Most of the studies followed the approach of decomposing the required algorithm into steps for parallel execution using NEON in the SIMD architecture. This method has the disadvantage as most of the SIMD architectures (including NEON) do not support the propagation carries between data elements that are processed in parallel, especially when the operands are represented in a non-redundant form (with sufficient radix).Due to this limitation, the paper in [8] provided a reduced radix representation method (the redundant form) to facilitate the handling of the carry propagations. However, as indicated in [25], this approach can lead to more intermediate products than the number of required operations. The reduced radix representation (the redundant representation) requires more multiplications than the canonical representation (the non-redundant representation). Several methods were proposed in [4, 25] to improve the performance by parallelizing parts in an operation with operands in the full radix form.

Another trend in applying NEON is to perform two operations in parallel on the finite fields. It is not necessary to handle the extra carry propagation during data parallelization. In [20], the authors presented the application of the NEON component in the dual multiplication to improve the efficiency of the Montgomery multiplication. However, the method in [20] used a lot of instructions to read and write data between the internal memory and NEON. The computational overhead for reading and writing data between the memory and the NEON component is often quite large. This problem has not been carefully considered in [20].

The studies in [4, 20, 25] focused on improving the Montgomery multiplication. This multiplication is said to be best for multiplying large numbers in a finite field. The multiplication of large numbers is more suitable for exponentiation operations (e.g, in RSA cryptosystems), but it is less useful for simple multiplications in ECC cryptosystems. The reason is that the Montgomery multiplication requires radix conversions. Radix conversions are often computationally expensive. Studies in [19] and [18] presented the possibility of improving the multiplication by decomposing the algorithm into parallel steps in a multiplication. The algorithm combined the Karatsuba multiplication with the NEON component in the SIMD architecture. However, the presented method only focused on the multiplication in the binary fields $GF(2)$, and did not deal with the carry propagation.

3 Elliptic Curves on Finite Fields

For the sake of clarity, this section presents a brief overview of the elliptic curves over finite fields. There are three basic coordinate forms that are commonly used. They are: 1) relative coordinates (Affine coordinates), 2) projective coordinates, and 3) compression coordinates. However, the relative coordinates and the projective coordinates are more commonly used [10, 14].

3.1 Affine Coordinates

Let $\mathbb{F}_p$ be a finite field with a prime number p, (E) be the elliptic curve over $\mathbb{F}_p$. A finite point P on (E) is defined by two elements, x and y, in $GF(p)$ that satisfy the equation of the curve:

$$(E): \ y^2 = x^3 + ax + b; \ a, \ b \in \mathbb{F}_q, 4a^3 + 27b^2 \neq 0 \ (mod \ p).$$

x and y are called the relative coordinates (the Affine coordinate) of the point P. The point at infinity ∞ has no Affine coordinates. For the purpose of calculations, ∞ is often represented by a pair of coefficients (x, y) that do not belong to (E).

The grouping rule is often used to express the relationship between the points in a finite field, and is defined as follows.

Grouping Rule: Let E be an elliptic curve defined on the field $\mathbb{F}_p$ by the equation $y^2 = x^3 + Ax + B$ with $A, B \in \mathbb{F}_p$ and $4A^3 + 27B^2 \neq 0 \ mod \ p$. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points on E with $P_1, P_2 \neq \infty$. The definition of $P_3 = (x_3, y_3)$ with $P_1 + P_2 = P_3$ is as follows:

1) If $x_1 \neq x_2$ then
$$x_3 = m^2 - x_1 - x_2, y_3 = m(x_1 - x_3) - y_1,$$
where $m = (y_2 - y_1)/(x_2 - x_1)$.

2) If $x_1 = x_2$ but $y_1 \neq y_2$, then
$$P_1 + P_2 = \infty$$

3) If $P_1 = P_2$ and $y_1 \neq 0$, then
$$x_3 = m^2 - 2x_1, y_3 = m(x_1 - x_3) - y_1,$$
where $m = (3x_1^2 + A)/(2y_1)$

4) If $P_1 = P_2$ and $y_1 = 0$, then
$$P_1 + P_2 = \infty$$

3.2 Projective Coordinates

To avoid the division in the fields, we suggest to represent the points in projective coordinates (coordinates in the fraction form). There are two basic types of projective coordinates, namely, the standard projective coordinates and the Jacobian projective coordinates.

In the standard projective coordinates, a point is represented as (X, Y, Z) with $Z \neq 0$, which is equivalent to the Affine coordinate of $(X/Z, Y/Z)$. The standard projective elliptic curve equation will have the following form:

$$Y^2Z = X^3 + aXZ^2 + bZ^3$$

In the Jacobian coordinates, a point (X, Y, Z) is equivalent to the point $(X/Z^2, Y/Z^3)$ in the Affine coordinates. The equation of curve (E) has the following form:

$$Y^2 = X^3 + aXZ^4 + bZ^6$$

A point with the Jacobian projective coordinates can be converted to the Affine coordinates according to the formula as follows:

$$(X, Y, Z) \rightarrow (x = X/Z^2, y = Y/Z^3).$$

Conversely, we can convert a point in the Affine coordinates to the Jacobian projective coordinates according to the formula as follows:

$$(x, y) \rightarrow (X = x, Y = y, Z = 1).$$

For additions and doublings of points in the projective coordinates, there is no need to use the inversions (divisions). The formula for the addition and doubling of two points in the projective coordinates can be obtained by converting the points to the Affine coordinates. After that, we apply the corresponding formula in the Affine coordinates, and finally, we remove the denominator of the formula.

3.3 Point Addition

Given 2 points $P(x_1 : y_1 : z_1)$, $Q(x_2 : y_2 : z_2)$ on the curve E with $P, Q \neq \infty$, $P \neq \pm Q$, we define the point $P + Q = (x_3 : y_3 : z_3)$ as follows:

Let

$$\begin{array}{lll} a = x_1 z_2^2, & b = x_2 z_1^2, & c = y_1 z_2^3 \\ d = y_2 z_1^3, & e = b - a, & f = d - c \end{array}$$

Then,

$$\begin{cases} x_3 = -e^3 - 2a \cdot e + f^2 \\ y_3 = c \cdot e^3 + f(a \cdot e^2 - x_3) \\ z_3 = z_1 \cdot z_2 \cdot e \end{cases}$$

3.4 Point doubling

Given a point $P(x_1 : y_1 : z_1)$ on the curve E with $P \neq \infty$, we define the point $[2]P = (x_3 : y_3 : z_3)$ as follows: Let

$$w_1 = 4x_1 \cdot y_1^2, w_2 = 3x_1^2 + A \cdot z_1^4$$

We have

$$\begin{cases} x_3 = -2w_1 + w_2^2 \\ y_3 = -8y_1^4 + w_2(w_1 - x_3) \\ z_3 = 2y_1 z_1 \end{cases}$$

The number of operations performed to add and double points in the coordinates is shown in Table 1 as follows (where I is the cost of the inverse operation and M is the cost of the multiplication).

Table 1: Number of operations performed to add and double points

Coordinates	Common addition	Doubling
Affine	1I,2M	1I,2M
Standard projective coordinates $(X/Z, Y/Z)$	$13M$	$7M$
Jacobian projective coordinates $(X/Z^2, Y/Z^3)$	$14M$	$5M$

3.5 Scalar Multiplications on Elliptic Curve

Scalar multiplications can be performed with several algorithms including:

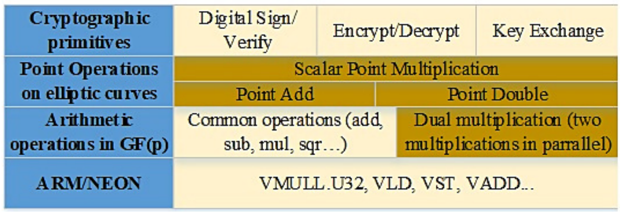
1) right-to-left binary algorithm,
2) NAF (Non-Adjacent Form) algorithm,
3) NAF algorithm with sliding window.

4 A Method for Improving the Efficiency of Point Arithmetic Operations

In this section, we propose a method for improving the efficiency of the point additions and doublings on the Elliptic curve $E(\mathbb{F}_p)$ using dual multiplications in the prime field $\mathbb{F}_p$. The proposed method focuses on leveraging the SIMD hardware features of ARM processors, especially of ARMv7 and ARMv8 [1] with the integrated NEON component to speed up arithmetic operations.

4.1 The Implementation Model

The model to perform multiplications is depicted in Figure 1. The basis of the model comprises: 1) the use of an available large number library, and 2) the implementation of parallel instructions.



Cryptographic primitives	Digital Sign/ Verify	Encrypt/Decrypt	Key Exchange
Point Operations on elliptic curves	Scalar Point Multiplication		
	Point Add	Point Double	
Arithmetic operations in GF(p)	Common operations (add, sub, mul, sqr...)	Dual multiplication (two multiplications in parrallel)	
ARM/NEON	VMULL.U32, VLD, VST, VADD...		

Figure 1: The implementation model

Most available large number libraries (e.g., RELIC, OpenSSL, MIRACL, GMP) represent large numbers in a non-redundant form with the radix 2^{32} or 2^{64} to match the basic multiplication support of the processors. The execution of two parallel multiplications has the advantage that we do not need to deal with the additional carry propagations in the case of integrating the proposed algorithms into the existing library. At the $GF(p)$ layer of the arithmetic operations, we propose to construct three dual multiplications that have the input operand of 256-bit, 384-bit, and 521-bit, respectively. At the next step, we integrate the dual multiplications into the point additions and point doublings by reorganizing the steps of the algorithm to build the pairs of the multiplications and squarings. Finally, we apply the NAF-based scalar multiplication algorithm to perform the scalar multiplications.

4.2 The Multiplication of Large Integers on $GF(p)$

Two basic methods are commonly used for the multiplication of large integers on $GF(p)$: 1) the operand scanning method, and 2) the product scanning method. These methods differ in the way they handle the operands, and in the number of instructions for loading and storing the data needed in the calculation. For describing the mentioned methods, we use the following notations. Let A and B be two large integers of m-bit length stored in the array

$$A = (A[s-1], ..., A[2], A[1], A[0])$$

$$\text{and } B = (B[s-1], ..., B[2], B[1], B[0]).$$

We denote with w the number of bits of the word, which is usually chosen according to the processor types (e.g., 8, 16, 32, 64 bit). We denote with $s = \lceil m/w \rceil$ the number of words for representing the integers A and B. The result of the multiplication, i.e., $C = A * B$, is represented using the array

$$C = (C[2s-1], ..., C[2], C[1], C[0]).$$

4.2.1 Operand Scanning Method

The operand scanning method is the simplest method (also known as the common multiplication [21]) for performing multiplications of large numbers that have two operands A and B with s words. The method is implemented through two nested loops: the outer loop (the i loop) is for loading the values $A[i]$, while the inner loop (the j loop) is for loading the values B[j] and multiplying by $A[i]$, where $j = 0, ..., s-1$. The partial products are accumulated into the intermediate result column $C[i+j]$ along with the carry of the previous column. The formula for calculating the product of the components is as follows:

$$Carry, C[i+j]) = C[i+j] + A[i] * B[j] + Carry.$$

Figure 2 illustrates how to calculate the component products. The method uses a row-wise view, where the flow of computation is in the direction of the arrows.

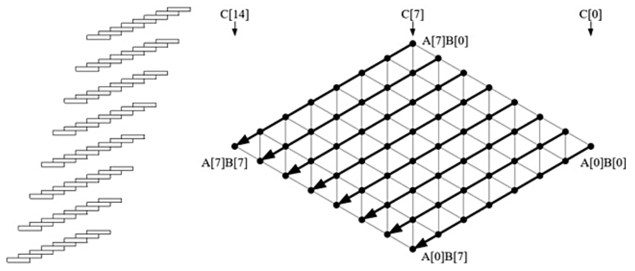


Figure 2: Multiplication of large numbers using the operand scanning method

The number of instructions for loading and saving data of this method is determined as follows:

- In each row: The number of load instructions in each inner loop is $2s$ (for loading data $B[j], C[i+j]$). The number of save instructions in each inner loop is s (to store $C[i+j]$). Thus, the total number of instructions in each row is $3s$.

- The outer loop needs s operands $A[i]$ for loading, and contains s values for carry, i.e., $C[i+s] = carry$. Thus, the total number of instructions is $2s$.

Therefore, the total cost of the operand scanning method is $2s^2+s$ for loading data, and is s^2+s for storing data, respectively. In total, this method takes $3s^2+2s$ instructions for loading and storing data. It is difficult to implement the parallelization of the algorithm, because the data depends on each other in a row-wise manner.

4.2.2 Product Scanning Method

The product scanning method implements the multiplications of large integers based on column-wise manner [6]. This method has the advantage of reducing the number of memory accesses. In cryptography, the number of columns to multiply does not exceed 2^w. We have: $s < 2^{3w}/2^{2w} = 2^w$, where w denotes the bit length of a word. Since the accumulator has a size of 3 words, it can contain the sum of all component multiplications of a column without having to contain intermediate results.

Figure 3 describes how to perform the product scanning multiplication. First, all operands of each column are multiplied by each other and their products are cumulatively added (i.e., using a cumulative multiplication method). After processing a column, the first word of the accumulator stored in the memory is a part of the final result. Therefore, no intermediate results are need for saving or loading in the algorithm. Furthermore, handling the carry propagations is quite easy, because the carries will be added to the result of the next columns. In addition, only five registers are required to perform the multiplication: two registers for storing the input operands, three registers for serving as the accumulators. This method is very suitable for the devices with limited resources. The formula for calculating products in the product scanning method is as follows:

$$C = A * B = \sum_{t=0}^{2s-2} \Big(\sum_{i+j=t, 0\leq i,j\leq s-1} A_i * B_j \Big) W^t$$

where $W = 2^w$ is called the radix.

The rhombus in Figure 3 represents the process of calculating component products in the column-wise manner instead of the row-wise manner in the operand scanning method. In the product scanning method, only one storage operation is needed for storing the word of the final result. The cost of the entire multiplication is as follows:

- Because the outer loop has a size of $2s$ and the inner loop changes from 0 to s, the number of instructions for loading data (i.e., for loading $A[i]$, and $B[j]$ in each loop) is $2s^2$.

- The number of instructions for saving data is $2s$ (each step of the outer loop only needs to save one value).

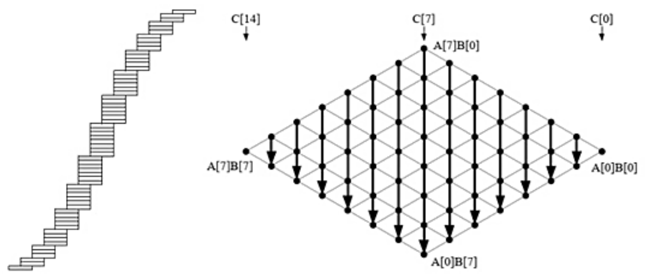


Figure 3: Multiplication of large numbers using the product scanning method

- The total number of instructions for loading and saving data is $2s^2 + 2s$.

The following table summarizes the cost of each method.

Table 2: Comparison of computational costs between multiplication methods

Methods	Number of instructions for loading data	Number of instructions for saving data	Total number of instructions
Operand scanning	$2s^2 + s$	$s^2 + s$	$3s^2 + 2s$
Product scanning	$2s^2$	$2s$	$2s^2 + 2s$

4.3 Parallelization of Two Multiplications on the Field GF(p)

The NEON component has sixteen 128-bit registers on the ARMv7 processors. Thus, we can directly implement the operand scanning method for multiplications (i.e., the common multiplications) with sizes of 256-bit and 384-bit. For 521-bit multiplications, we suggest using the conventional Karatsuba algorithm for the operations based on the combination of implementations in C and in NEON, since there are not enough registers to directly implement on NEON. On the ARMv8 processors, the NEON component has more registers (i.e., thirty-two 128-bit registers). However, we are able to use the same algorithm for both platforms, ARMv7 and ARMv8.

The modulo calculation algorithm uses the primitive algorithm as presented in [10]. The SIMD architecture has the feature of supporting two 32-bit multiplications using a single instruction. We apply this operation in Algorithm 1 as follow.

Figure 4 shows the parallel execution of two multiplications. The NEON instructions execute inside the loop of Step 4 in Algorithm 1 as shown in Figure 5.

Algorithm 1 Parallel multiplication of two multiplications on the field $GF(p)$

Input: $A = (A_{s-1}, ..., A_1, A_0)$, $B = (B_{s-1}, ..., B_1, B_0)$ and $C = (C_{s-1}, ..., C_1, C_0)$, $D = (D_{s-1}, ..., D_2, D_1, D_0)$.
Output: $M = (M_{2s-1}, \ldots, M_1, M_0) = A \cdot B$ and $N = (N_{2s-1}, .., N_1, N_0) = C \cdot D$.

```
1:  $M = 0,\ N = 0$
2:  for i=0 to s-1 do
3:      $T_1 = 0,\ T_2 = 0$
4:      for j=0 to s-1 do
5:          $(T_1, S_1) = M_{i+j} + A_i \cdot B_j + T_1,\ (T_2, S_2) = M_{i+j} + C_i \cdot D_j + T_2$
6:          $M_{i+j} = S_1,\ N_{i+j} = S_2$
7:      end for
8:      $M_{i+s} = T_1, N_{i+s} = T_2$
9:  end for
10: return $(M, N)$
11: End
```

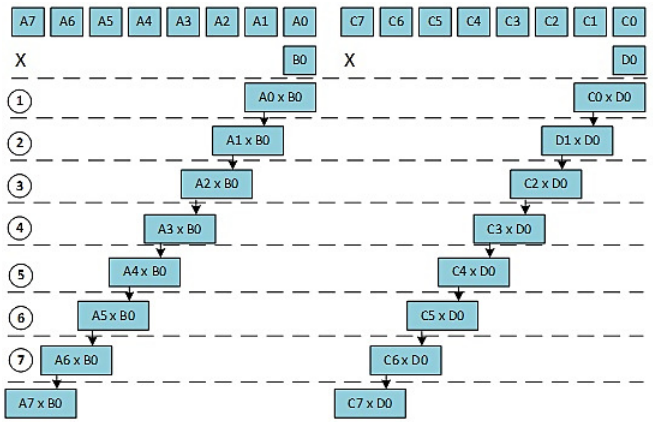


Figure 4: Parallel multiplication of two multiplications

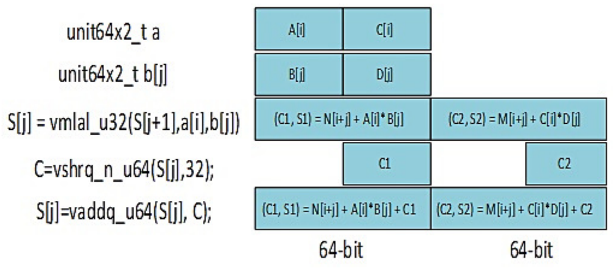

unit64x2_t a
unit64x2_t b[j]
S[j] = vmlal_u32(S[j+1],a[i],b[j])
C=vshrq_n_u64(S[j],32);
S[j]=vaddq_u64(S[j], C);
A[i]
C[i]
B[j]
D[j]
(C1, S1) = N[i+j] + A[i]*B[j]
(C2, S2) = M[i+j] + C[i]*D[j]
C1
C2
(C1, S1) = N[i+j] + A[i]*B[j] + C1
(C2, S2) = M[i+j] + C[i]*D[j] + C2
64-bit
64-bit

Figure 5: Calculations in the j-th loop of Algorithm 1 using NEON

As indicated in [21], the values (T_1, S_1) and (T_2, S_2) are represented within two words (for this case the radix is 2^{32}). Because it is able to perform two multiplications simultaneously, the NEON instructions are useful to speed up the computations of the arithmetic algorithms. However, it is expensive for loading and storing data between the NEON registers and the internal memory. Loading and storing instructions are used much in the existing studies, as we pointed out in the section 2 of this paper. Such instructions can result in a high computational overhead. For example, data variables are continuously loaded into the NEON registers and removed from memory in two loops as shown in Algorithm 3 in the paper [20]. The mentioned method cannot achieve the Pipeline mechanism, and the performance of the parallel multiplication in NEON is significantly slowed down.

Therefore, this paper proposes to minimize the number of instructions for loading and storing data for leveraging the advantages of parallel computations in NEON. In this manner, our method differs from existing methods, e.g. the method presented in [20]. The method for restricting the number of instruction is described in detail as below.

4.3.1 Instruction Restriction for 256-bit and 384-bit Multiplications

Since it takes a long time for reading and writing data between the memory and the NEON registers (using vld and vst instructions), we read all inputs from the memory, and then apply Algorithm 1 for the computation. The result is written back from the NEON registers to the memory. This manner can minimize the time of reading and writing data between the memory and the NEON registers.

- For 256-bit multiplications: The 256-bit input operands are stored in an 8-word memory array, each word consists of 32 bits. Eleven 128-bit registers are required. Among them, 9 registers are for storing intermediate products and 2 registers are for storing temporary variables (e.g. carries, variables for getting the lower part of the data). In addition, 10 registers of 64-bit are needed, where 9 registers are for storing input terms and one register is for storing temporary variables.

- For 384-bit multiplications: The 384-bit input operands are stored in a 12-word memory array, each word consists of 32 bits. Fifteen 128-bit registers are required, where 13 registers are for storing intermediate products and 2 registers are for storing temporary variables. In addition, fourteen 64-bit registers are needed, where 13 registers are for storing the input terms and one register is for storing temporary variables.

4.3.2 Evaluating the Number of Instructions for Loading and Storing Data

Algorithm 3 in the study [20] presented the instructions for loading and storing data using the $neon\_dual\_mac2$ function. Because the dual multiplication is performed, the number of instructions is twice as many as the one in the operand scanning multiplication method. Thus, the total number of instructions for accessing the memory is required as $2 \times (3s^2 + 2s)$.

In our algorithm, i.e. Algorithm 1, the loop i and j only require to load the operands A, B, C, D, and then to write the final result (M, N) from the NEON registers to the memory. Thus, it takes 4s instructions for loading data and $2 \times 2s$ instructions for storing data.

The following table compares the number of data loading and saving instructions between our new proposed algorithm and Algorithm 3 presented in the study [20].

Table 3: Comparison of the number of instructions for data loading and saving between two algorithms

Methods	Number of instructions for loading data	Number of instructions for saving data	Total number of instructions for accessing memory
Algorithm 3 [20]	$2(2s^2 + 2)$	$2(s^2 + s)$	$6s^2 + 4s$
Algorithm 1 (256-bit and 384-bit multiplications)	$4s$	$4s$	$8s$

4.3.3 Instruction Restriction for 521-bit Multiplications

In the case of 521-bit multiplications, the 521-bit input operand is stored in a memory array of 17 words, each word consists of 32 bits. Twenty 128-bit registers are needed. However, the ARMv7's NEON component only has sixteen 128-bit registers, i.e., it has not enough registers for implementing the parallel multiplications. Therefore, we propose a combination mechanism that applies Karatsuba algorithm to perform 521-bit multiplications with the C/NEON programming language as follows.

Step 1. Implementing two additional multipliers, i.e., a dual multiplier for two 288-bit (9 words) operands and a dual multiplier for two 320-bit (10 words) operands. The procedure is similar to that for the dual multiplications of 256-bit operands as presented above.

Step 2. Applying the conventional Karatsuba algorithm with one level. The 1-level Karatsuba method is based on the paper [26]. We split the 521-bit operand (stored in seventeen 32-bit words) into 2 parts: The first part has 8 words and the second part has 9 words. Two common approaches can be used for implementing the Karatsuba algorithm, namely the additive and the subtractive algorithms. Suppose that, we want to multiply two pairs of operands, $M = A \cdot B$ and $N = C \cdot D$, where $A = A_H \cdot 2^{256} + A_L$, $B = B_H \cdot 2^{256} + B_L$ and $C = C_H \cdot 2^{256} + C_L$, $D = D_H \cdot 2^{256} + D_L$. The dual multiplications

$M = A \cdot B$ and $N = C \cdot D$ are calculated according to the following addition formula:

$$A_H \cdot B_H \cdot 2^{521} + [(A_H + A_L)(B_H + B_L) - A_H \cdot B_H - A_L \cdot B_L] \cdot 2^{256} + A_L \cdot B_L$$

$$C_H \cdot D_H \cdot 2^{521} + [(C_H + D_L)(C_H + D_L) - C_H \cdot D_H - C_L \cdot D_L] \cdot 2^{256} + C_L \cdot D_L.$$

The operands A_L, B_L, C_L, D_L have a length of 256-bits (8 words). Therefore, the pairs of multiplications $A_L \cdot B_L$ and $C_L \cdot D_L$ are performed using the 8-word dual multiplication algorithm that is inherited from the dual multiplier with 256-bit operand length. The operands A_H, B_H, C_H, D_H have a length of 288-bits (9 words). Thus,the pairs of multiplications $A_H \cdot B_H$ and $C_H \cdot D_H$ are performed using the 9-word dual multiplication algorithm by a dual multiplier with 288-bit operand length. The operands $(A_H + A_L)$, $(B_H + B_L)$, $(C_H + C_L)$, $(D_H + D_L)$ are 320-bits (10 words) long. Thus, the pairs the multiplications $(A_H + A_L)(B_H + B_L)$ and $(C_H + C_L)(D_H + D_L)$ are performed using the 10-word dual multiplication algorithm that uses a dual multiplier with 320-bit operand length.

Algorithm 2 Dual Karatsuba multiplications for 521-bit

Input: Four 17-word operands $A = A_H||A_L, B = B_H||B_L$ and $C = C_H||C_L, D = D_H||D_L$ (each word consists of 32 bits).

Output: $M = A \cdot B$ and $N = C \cdot D$ with the length of 34 words (each word consists of 32 bits).

```
1: $M_L = A_L \cdot B_L$ and $N_L = C_L \cdot D_L$ {NEON, 256bit}
2: $M_H = A_H \cdot B_H$ and $N_H = C_H \cdot D_H$ {NEON, 288bit}
3: $A_{HL} = (A_H + A_L)$, $B_{HL} = (B_H + B_L)$ {C, 320bit}
4: $C_{HL} = (C_H + C_L)$, $D_{HL} = (D_H + D_L)$ {C, 320bit}
5: $M_M = A_{HL} \cdot B_{HL}$ and $N_M = C_{HL} \cdot D_{HL}$ {NEON, 320bit}
6: $M = M_H \cdot 2^{521} + (M_M - M_H - M_L) \cdot 2^{256} + M_L$ {C}
7: $N = N_H \cdot 2^{521} + (N_M - N_H - N_L) \cdot 2^{256} + N_L$ {C}
```

4.4 Point Arithmetic Algorithms on Elliptic Curves

As presented in the study [2], the algorithms for adding and doubling points on the Jacobi coordinates use the multiplication and squaring operations. As we analyzed in the previous section, we can use the dual multiplication algorithms to perform the pairs of multiplications in parallel for the pairs of multiplications on F_p, whose data do not depend on each other... This principle can also be applied to the squarings. To simplify the implementation process, we do not implement the dual squarings (i.e., parallelizing two squarings). Instead, we use the dual multiplication to perform two squarings in parallel. In the following section, the paper presents several improvements for the point addition and doubling algorithms that originally presented in the paper [2].

4.4.1 Point Addition Algorithm

Algorithm 3 presented in the paper [2] is a point addition algorithm using the "add-2007-bl" formula with the sequential multiplication.

Algorithm 3 Point adding using the sequential multiplication algorithm [2]

Input: $P_1 = (X_1, Y_1, Z_1)$, $P_2 = (X_2, Y_2, Z_2)$ represented in Jacobi coordinates.

Output: $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3)$.

```
1: $T_1 = Z_1^2$
2: $T_2 = Z_2^2$
3: $U_1 = X_1 \cdot T_1$
4: $U_2 = X_2 \cdot T_2$
5: $S_1 = Y_1 \cdot Z_2 \cdot T_2$
6: $S_2 = Y_2 \cdot Z_1 \cdot T_1$
7: $H = U_2 - U_1$
8: $I = (2 \cdot H)^2$
9: $J = H \cdot I$
10: $R = 2 \cdot (S_2 - S_1)$
11: $V = U_1 \cdot I$
12: $X_3 = R^2 - J - 2 \cdot V$
13: $Y_3 = R \cdot (V - X_3) - 2 \cdot S_1 \cdot J$
14: $Z_3 = ((Z_1 + Z_2)^2 - T_1 - T_2) \cdot H$
```

Since the point addition algorithm on the Jacobi coordinates can be decomposed into independent operations, we can apply the dual multiplication algorithm for this point addition algorithm. Thus, we propose to improve Algorithm 3 of the paper [2] by our Algorithm 4 using the dual multiplication that allows the parallel execution as follow. In Algorithm 4, the multiplications and squarings are organized into pairs of multiplications and pairs of squarings whose data are independent of each other. We can see that Algorithm 4 is equivalent to Algorithm 3, except that Algorithm 4 leverages the parallel execution.

4.4.2 Point Doubling Algorithm

Algorithm 5 described below is a point doubling algorithm using the formula "dbl-2001-b" as presented in the paper [2], which use the sequential multiplication.

Similar to point addition, the point doubling on the Jacobi coordinates can be decomposed into independent operations. Therefore, we can apply the dual multiplication algorithm for the point doubling algorithm. We propose to improve Algorithm 5 of the paper [2] by our Algorithm 6 using the double multiplication that allows the parallel execution as follow.

From the implementation of the steps, we can see that our Algorithm 6 is equivalent to Algorithm 5 of [2]. However, our Algorithm 6 differs from Algorithm 5 by leveraging the parallel execution.

Algorithm 4 Adding two points using SIMD-based dual multiplication algorithm

Input: $P_1 = (X_1, Y_1, Z_1)$, $P_2 = (X_2, Y_2, Z_2)$ represented in Jacobi coordinates.
Output: $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3)$.

```
1: $T_1 = Z_1^2$, $T_2 = Z_2^2$    {NEON}
2: $U_1 = X_1 \cdot T_1$, $U_2 = X_2 \cdot T_2$    {NEON}
3: $S_1 = Z_2 \cdot T_2$, $S_2 = Z_1 \cot T_1$    {NEON}
4: $S_1 = Y_1 \cdot S_1, S_2 = Y_2 \cdot S_2$    {NEON}
5: $H = U_2 - U_1$    {C}
6: $I = 2 \cdot H$    {C}
7: $I = I^2, T_3 = R^2$    {NEON}
8: $J = H \cdot I$, $V = U_1 \cdot I$    {NEON}
9: $R = 2 \cdot (S_2 - S_1)$    {C}
10: $X_3 = T_3 - J - 2 \cdot V$    {C}
11: $T_3 = V - X_3$    {C}
12: $T_3 = R \cdot T_3$, $T_4 = S_1 \cdot J$    {NEON}
13: $Y_3 = T_3 - 2 \cdot T_4$    {C}
14: $Z_3 = ((Z_1 + Z_2)^2 - T_1 - T_2) \cdot H$ {C}
```

Algorithm 5 Point doubling using the sequential multiplication algorithm [2]

Input: $P_1 = (X_1, Y_1, Z_1)$ represented in Jacobi coordinates.
Output: $P_3 = 2 \cdot P_1 = (X_3, Y_3, Z_3)$.

```
1: $T_0 = delta = Z_1^2$
2: $T_1 = gamma = Y_1^2$
3: $T_2 = beta = X_1 \cdot T_1$
4: $T_3 = X_1 - T_0$
5: $T_4 = X_1 + T_0$
6: $T_3 = alpha = 3 \cdot T_3 \cdot T_4$
7: $X_3 = T_3^2 - 8 \cdot T_2$
8: $Z_3 = (Y_1 + Z_1)^2 - T_1 - T_0$
9: $Y_3 = T_3 \cdot (4 \cdot T_2 - X_3) - 8 \cdot T_1^2$
```

Algorithm 6 Point doubling using the sequential multiplication algorithm [2]

Input: $P_1 = (X_1, Y_1, Z_1)$ represented in Jacobi coordinates.
Output: $P_3 = 2 \cdot P_1 = (X_3, Y_3, Z_3)$.

```
1: $T_0 = delta = Z_1^2, T_1 = gamma = Y_1^2$    {NEON}
2: $T_3 = X_1 - T_0$    {C}
3: $T_4 = X_1 + T_0$    {C}
4: $T_2 = beta = X_1 \cdot T_1$, $T_3 = T_3 \cdot T_4$    {NEON}
5: $T_3 = alpha = 3 \cdot T_3$    {C}
6: $T_5 = Y_1 + Z_1$    {C}
7: $T_4 = T_3^2$; $T_5 = T_5^2$    {NEON}
8: $X_3 = T_4 - 8 \cdot T_2$    {C}
9: $Z_3 = T_5 - T_1 - T_0$    {C}
10: $T_4 = 4 \cdot T_2 - X_3$    {C}
11: $T_4 = T_3 \cdot T_4$, $T_5 = T_1^2$    {NEON}
12: $Y_3 = T_4 - 8 \cdot T_5$    {C}
```

4.4.3 Comparison of the proposed algorithms with the original algorithms

For the comparison of our parallel algorithm with the sequential algorithm [2], we use the following notations. We denote M, S as the cost of the conventional multiplication and squaring, and Mt, St as the cost of the dual multiplication according to the proposed algorithms.

The cost of the point addition according to the sequential model [2] (i.e., Algorithm 3) is $11M + 5S$, while the cost of our proposed addition algorithm (i.e., Algorithm 4) is $5Mt + 2St + 1M + 1S$. Since $Mt < 2M$ and $Mt < 2S$, we can see that the cost of the proposed algorithm (i.e., Algorithm 4) is more efficient than the cost of the point addition algorithm presented in [2] (i.e, the Algorithm 3).

Table 4: Comparison of the costs of the algorithms

Algorithm	Cost of the point operation	
	Cost of the point addition	Cost of the point doubling
Sequential algorithm [2]	$11M + 5S$	$3M + 5S$
Our proposed parallel algorithm	$5Mt + 2St + 1M + 1S$	$4Mt$

For the point doubling algorithms, the cost of the sequential point doubling [2] (i.e., Algorithm 5) is 3M+5S, while the cost of the proposed point doubling (i.e., Algorithm 6) is 4Mt. Thus, the cost of the proposed algorithm (i.e., Algorithm 6) is more efficient than the cost of the point doubling algorithm presented in [2] (i.e., Algorithm 5).

Table 4 shows the comparison of the costs of the mentioned algorithms.

4.4.4 Scalar Point Multiplication

For the scalar point multiplication, we use the sliding window NAF algorithm to multiply a positive integer by a point. In this scalar point multiplication, our algorithm uses the point addition and the point doubling according to Algorithms 4 and 6 based on the SIMD dual multiplication.

5 Experiments and Evaluations

5.1 Experiment Settings

The following tests are performed on ARMv7 and ARMv8 processors with the integrated NEON component. The NEON [1] instructions are used to perform two multiplications in parallel including:

- $vmlal\_u32$ (for multiplying and accumulating unsigned integers in the vector form):
  For $Q0 = vmlal\_u32(Q0, D2, D3[0])$, we calculate

$$D1 = D1 + D2[1] \times D3[0] \text{ and}$$

$$D0 = D0 + D2[0] \times D3[0].$$

- *vshrq_n_u*64 (to right shift the vector by n bits)): For $Q0 = vshrq\_n\_u64(Q0, 32)$, we calculate

$$D0 = D0 \gg 32 \text{ and } D1 = D1 \gg 32.$$

- *vaddq_u*64 (to perform addition in vector form): For $Q0 = vaddq\_u64(Q0, Q1)$, we calculate

$$D0 = D0 + D2 \text{ and } D1 = D1 + D3.$$

- *vandq_u*64 (to perform AND operation in vector form): For $Q0 = vandq\_u64(Q0, Q1)$, we calculate

$$D0 = D0 \& D2 \text{ and } D1 = D1 \& D3.$$

- *vmovn_u*64 (to transfer vector data in narrow mode) For $D0 = vmovn\_u64(Q1)$, we calculate

$$D0[0] = D2[0] \text{ and } D0[1] = D3[0].$$

- *vld1_u*32 (to load data from memory into registers in NEON) For $D0 = vld1\_u32([N])$, we calculate

$$D0[0] = [N] \text{ and } D0[1] = [N + 1].$$

- *vget_lane_u*32 (to copy data from NEON registers to memory) For $R1 = vget\_lane\_u32(D0, 0)$, we calculate

$$R1 = D0[0].$$

For $R2 = vget\_lane\_u32(D0, 1)$, we calculate

$$R2 = D0[1].$$

Based on the above instructions, we implement the dual multiplication algorithm, the point addition algorithm, and the point doubling algorithm as described in the section 4 based on the RELIC cryptographic library version 0.5.0. The RELIC is a cryptographic library that is considered to have a fairly fast execution speed among other open source cryptographic libraries such as OpenSSL and GMP. Next, we performed experiments and evaluated the results between the library that integrates the proposed improvements and the original library that uses the default algorithm. To measure the time of an operation, we take the average number of 100 executions of that operation.

5.2 Experiments on ARMv7

The following experiment results are performed on the hardware platform: a Xilinx Zynq Kit with the ARMv7 1.3 GHz processor running an embedded Linux operating system. The development tool used for compiling the program is arm-xilinx-linux-gnueabi-gcc version 4.8.3. We performed tests for the proposed multiplication algorithm on three curves: Curve NIST-P256, Curve NIST-P384, and Curve NIST-P521.

The experiment results with ARMv7 are presented in Tables 5, 6, and 7. Table 5 shows the results of performing the arithmetic operations on F_p. Table 6 presents the results for performing the point arithmetic operations on the curve $E(F_p)$. Table 7 depicts the results with the cryptographic primitive functions based on the $E(F_p)$ curves.

Table 5 shows the comparative evaluation test between the time for performing a dual multiplication and the time for performing two multiplications (Mul) or two squarings (Sqr) sequentially (i.e., the default operation using the RELIC library). The last column indicates the ratio (i.e., the efficiency). As shown in Table 5, the proposed algorithm is faster than the default algorithms using the RELIC library, namely 30% and 20% faster for the multiplications and squarings, repectively.

Table 6 presents the comparative evaluation test between the execution time of the point addition (Add), the point doubling (Dbl) and the scalar point multiplication ($k \cdot P$) of the proposed algorithm with that of the default operations using the RELIC library. The results in the last column show that, the proposed algorithm (i.e., using the NEON component) provides results of 20% to 30% faster than the default algorithm using the RELIC library.

Table 7 shows the performance evaluation for two cryptographic protocols (ECDH, ECDSA) using our proposed algorithm (i.e., the algorithm based on NEON) and the algorithm using the original RELIC library. On the ARMv7 platform, the performance of the proposed algorithms for ECDH and ECDSA protocols increases by 10% to 20% compared to the original algorithms.

5.3 Experiments on ARMv8

The following experiment results are performed on the hardware platform: a NXP IMX8M Kit with the ARMv8 1.5 GHz processor running an Android 10 operating system. We use the Android development tool NDK (Standalone toolchains) to compile the programs with GCC compiler version 6.3. We performed tests for the proposed multiplication algorithm on three curves: Curve NIST-P256, Curve NIST-P384, and Curve NIST-P521.

The experiment results with ARMv8 are presented in Tables 8, 9, and 10. Table 8 shows the results for performing the arithmetic operations on F_p. Table 9 presents the results for performing the point arithmetic operations on the curve $E(F_p)$. Table 10 indicates the results with the cryptographic primitive functions based on the $E(F_p)$ curves.

Table 8 shows the comparative evaluation test between the time for performing a dual multiplication and the time for performing two multiplications (Mul) or two squarings (Sqr) sequentially (i.e., the default operation using the RELIC library). The results in the last column show

Table 5: Arithmetic operations on F_p with ARMv7

Bit Length	Arithmetic operations on F_p	Time (10^3 nanoseconds)		Ratio
		Dual multiplication	Sequential multiplication	
256	Mul	4.9	6.9	0.71
	Sqr	4.9	6.3	0.78
384	Mul	9.2	13.5	0.68
	Sqr	9.2	13.0	0.71
521	Mul	19.6	28.6	0.69
	Sqr	19.6	28	0.70

Table 6: Point arithmetic operations on the curve $E(F_p)$ with ARMv7

Bit Length	Arithmetic operations on F_p	Time (10^3 nanoseconds)		Ratio
		Dual multiplication	Sequential multiplication	
256	Add	43	57.2	0.75
	Dbl	21.8	30.3	0.72
	$k \cdot P$	8742	10770	0.81
384	Add	81.5	107.8	0.76
	Dbl	40.6	57.3	0.71
	$k \cdot P$	23672	29476	0.80
521	Add	176.5	238.5	0.74
	Dbl	87.8	125.2	0.70
	$k \cdot P$	68243	85689	0.80

Table 7: $E(F_p)$-based cryptographic primitives with ARMv7

Bit Length	Protocols based on $E(F_p)$	Time (10^3 nanoseconds)		Ratio
		SIMD	Sequential multiplication	
256	ECDH	9.0	11.2	0.8
	ECDSA Sig	4.9	5.3	0.92
	ECDSA Ver	12.6	14.7	0.86
384	ECDH	23.4	29.3	0.80
	ECDSA Sig	12.6	13.6	0.93
	ECDSA Ver	32.5	37.3	0.87
521	ECDH	69.4	86.9	0.80
	ECDSA Sig	36.1	39.3	0.92
	ECDSA Ver	74.8	85.0	0.88

Table 8: Arithmetic operations on F_p with ARMv8

Bit Length	Arithmetic operations on F_p	Time (10^3 nanoseconds)		Ratio
		Dual multiplication	Sequential multiplication	
256	Mul	1.8	2.9	0.62
	Sqr	1.8	2.4	0.75
384	Mul	3.6	6.1	0.60
	Sqr	3.6	5.1	0.71
521	Mul	8.4	13.8	0.61
	Sqr	8.4	11.0	0.76

that, the proposed algorithm is faster than the default algorithm using the RELIC library, namely 40% and 25% faster for the multiplications and squarings, repectively.

Table 5 presents the comparative evaluation test between the execution time of the point addition (Add), the point doubling (Dbl) and the scalar point multiplication ($k \cdot P$) of the proposed algorithm with that of the default operations using the RELIC library. The results in the last column show that, the proposed algorithm (i.e. the algorithm using the NEON component) is about 20% to 30% faster than the default algorithm using the RELIC library.

Table 9: Point arithmetic operations on the curve $E(F_p)$ with ARMv8

Bit Length	Arithmetic operations on F_p	Time (10^3 nanoseconds)		Ratio
		Dual multiplication	Sequential multiplication	
256	Add	16.2	22.8	0.71
	Dbl	8.4	12.2	0.69
	$k \cdot P$	3376	4149	0.81
384	Add	32.7	47.6	0.69
	Dbl	17.5	24.0	0.73
	$k \cdot P$	9645	12409	0.78
521	Add	74.9	100.8	0.74
	Dbl	37.6	50.3	0.75
	$k \cdot P$	28696	34825	0.82

Table 10 shows the evaluation of the performance of two cryptographic protocols (ECDH, ECDSA) using our proposed algorithm (based on NEON) and the original protocol implemented in the RELIC library. On the ARMv8 platform, the performance of the proposed algorithms for ECDH and ECDSA protocols increases by 10% to 20% compared to the original algorithms.

Table 10: $E(F_p)$-based cryptographic primitives with ARMv8

Bit Length	Protocols based on $E(F_p)$	Time (10^3 nanoseconds)		Ratio
		SIMD	Sequential multiplication	
256	ECDH	3.4	4.2	0.81
	ECDSA Sig	1.8	2.0	0.90
	ECDSA Ver	4.7	5.5	0.85
384	ECDH	9.7	12.6	0.77
	ECDSA Sig	5.1	5.8	0.88
	ECDSA Ver	13.0	16.0	0.81
521	ECDH	28.6	34.8	0.82
	ECDSA Sig	14.7	16.0	0.92
	ECDSA Ver	37.1	42.9	0.86

6 Conclusions

Techniques for improving the efficiency of the point arithmetic operations, especially the computational speed, is always a challenge for the elliptic curve cryptosystems using hardware platforms with low processing power. The traditional Karatsuba algorithm allows multiplying integers with low computational complexity and fast speed. However, it is difficult to implement the algorithm on processors with the limited performance. Several studies focused on reducing the number of operations, and thus, the cost of computation. The application of the SIMD architecture with the NEON component can provide the possibility for calculations in parallel. In addition, we can leverage the hardware characteristics to reduce the number of intermediate calculations, as well as combine the Karatsuba algorithm with advanded hardware platforms such as ARM. However, this approach has not yet been fully investigated.

This paper proposes a method for improving the efficiency of the arithmetic operations including the point addition, doubling and scalar point multiplication on elliptic curves using the ARM hardware characteristics and the NEON. In this method, we combine the operand scanning multiplication with the Karatsuba algorithm. The proposed method performs the parallel multiplication (the dual multiplications) together with the multiplication pairing to minimize the cost of data read/write operations between the memory and the NEON. Furthermore, we use an available large number library to speed up the computation. We proposed several algorithms to implement the method on ARMv7 and ARM v8 embedded processors. The algorithms are fully integrated into the computations of the EDCH and ECDSA protocols on the $GF(p)$ fields with sizes of 256, 384, and 521 bits. Experimental results shown that the efficiency of the proposed algorithms increases from 20% to 30% for the basic operations (including the point addition, doubling, and scalar point multiplication), and increases from 10% to 20% for the calculations in the ECDH and ECDSA protocols in comparison with the previous method.

Further extensions of this work can be the improved parallelization by combining the interleavedg ARM instructions and the SIMD instructions, or the adaption of the algorithms to the other ARM-Cortex processor families.

References

[1] ARM, "Series programmer's guide," Technical Report Cortex-A, 2012.

[2] D. J. Bernstein, T. Lange, and *et al.*, "Explicit-formulas database," Jan. 27, 2022. (`https://hyperelliptic.org/EFD/`)

[3] D. J. Bernstein and P. Schwabe, "Neon crypto," in *Proceedings of International Workshop on Cryptographic Hardware and Embedded Systems (CHES'12)*, pp. 320–339, Springer, 2012.

[4] J. W. Bos, P. L. Montgomery, D. Shumow, and G. M. Zaverucha, "Montgomery multiplication using vector instructions," in *Proceedings of Conference Selected Areas in Cryptography (SAC'13)*, pp. 471–489, Springer, 2013.

[5] D. Câmara, C. P. Gouvêa, J. López, and R. Dahab, "Fast software polynomial multiplication on arm processors using the neon engine," in *Proceedings of International Conference on Security Engineering and Intelligence Informatics*, pp. 137–154, Springer, 2013.

[6] P. G. Comba, "Exponentiation cryptosystems on the ibm pc," *IBM Systems Journal*, vol. 29, no. 4, pp. 526–538, 1990.

[7] M. S. Hwang, S. F. Tzeng, C. S. Tsai, "Generalization of proxy signature based on elliptic curves", *Computer Standards & Interfaces*, vol. 26, no. 2, pp. 73–84, 2004.

[8] Intel Corporation, "Using streaming simd extensions (SSE2) to perform big multiplications," Technical Report Application note AP-941, July 2000.

[9] A. Faz-Hernandez, P. Longa, and A. H. Sanchez, "Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves," in *Proceedings of RSA Conference: Topics in Cryptology CT-RSA*, pp. 1–27, Springer, 2014.

[10] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Berlin: Springer-Verlag, 2004.

[11] H. Cheng, J. Grosschaedl, J. Tian, P. B. Ronne, and P. Y. Ryan, "High-throughput elliptic curve cryptography using avx2 vector instructions," in *International Conference on Selected Areas in Cryptography (SAC'20)*, pp. 698–719, Springer, 2020.

[12] C. Y. Lee, C. C. Fan, J. Xie, and S. M. Yuan, "Efficient implementation of karatsuba algorithm based three-operand multiplication over binary extension field," *IEEE Access*, vol. 6, pp. 38234–38242, 2018.

[13] S. G. Liu, S. J. An, and Y. W. Du, "Efficient and secure elliptic curve scalar multiplication based on quadruple-and-add," *International Journal of Network Security*, vol. 23, no. 5, pp. 750–757, 2021.

[14] S. G. Liu, Y. Y. Hu, and L. Wei, "Elliptic curve scalar multiplication algorithm based on side channel atomic block over GF(2^m)," *International Journal of Network Security*, vol. 23, no. 6, pp. 1005–1011, 2021.

[15] Z. Liu, J. Groschdl, Z. Hu, K. Jrvinen, H. Wang, and I. Verbauwhede, "Elliptic curve cryptography with efficiently computable endomorphisms and its hardware implementations for the internet of things," *IEEE Transaction on Computers*, vol. 66, no. 5, p. 773–785, 2017.

[16] L. Kowada, R. Portugal, and C. M. H. Figueiredo, "Reversible karatsuba's algorithm," *Journal of Universal Computer Science*, vol. 12, no. 5, pp. 499–511, 2006.

[17] P. Longa, "Fourqneon: Faster elliptic curve scalar multiplications on arm processors," in *Proceedings of Conference Selected Areas in Cryptography (SAC'16)*, pp. 501–519, Springer, 2016.

[18] P. V. Luc, H. D. Hai, and L. D. Tan, "Multilayer multiplication in binary field on armv8 processors," in *Proceedings of IEEE International Conference on Advanced Technologies for Communications (ATC'20)*, Nhatrang, Vietnam, 2020.

[19] P. V. Luc, V. T. Linh, H. D. Hai, and L. D. Tan, "Fast binary field mutiplication on armv7 embedded processors," in *Proceedings of 4th IEEE International Conference on Recent Advances in Signal Processing, Telecommunications & Computing (SigTelCom'20)*, Hanoi, Vietnam, 2020.

[20] R. C. Marquez, A. J. C. Sarmiento, and S. Sánchez-Solano, "Speeding up elliptic curve arithmetic on arm processors using neon instructions," *RIELACy*, vol. 41, no. 3, pp. 1–20, 2020.

[21] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[22] National Institute of Standards and Technology (NIST), "DSS digital signature standard (DSS)," Technical Report Federal Information Processing Standards Publication 186-2, 2000.

[23] National Institute of Standards and Technology (NIST), "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography," Technical Report SP 800-56A Rev. 3, Apr. 2018.

[24] H. Seo, Z. Liu, J. Choi, and H. Kim, "Karatsuba-block-comb technique for elliptic curve cryptography over binary fields," *Journal of Security and Communication Networks*, vol. 8, no. 17, pp. 3121–3130, 2015.

[25] H. Seo, Z. Liu, J. Grobschadl, J. Choi, and H. Kim, "Montgomery modular multiplication on arm-neon revisited," in *Proceedings of International Conference on Information Security and Cryptology (ICISC'14)*, pp. 328–342, Springer, Cham, 2015.

[26] A. H. Sánchez and F. Rodríguez-Henríquez, "Neon implementation of an attribute based encryption scheme," in *Proceedings of International Conference on Applied Cryptography and Network Security, (ACNS'13)*, pp. 322–338, Springer, 2013.

[27] IEEE Standards, "Standard specifications for public key cryptography," Technical Report of IEEE 1363-2000, 2004.

[28] A. Weimerskirch and C. Paar, "Generalizations of the karatsuba algorithmfor efficient implementations," Technical Report of University of Ruhr, Bochum, Germany, 2003.

[29] C. C. Yang, T. Y. Chang, M. S. Hwang, "A new anonymous conference key distribution system based on the elliptic curve discrete logarithm problem", *Computer Standards and Interfaces*, vol. 25, no. 2, pp. 141–145, 2003.

Biography

Pham Van Luc received master's degree in cryptographic techniques in 2008. His research interests include Analysis and Design of Security Protocols, and Applied Cryptography.

Dang Hai Hoang, Dr.-Ing. (1999), Dr.-Ing.habil. (2002) in telematics and communication systems from Technical University of Ilmenau, Germany; Associate Professor at Posts and Telecommunications Institute of Technology, Hanoi, Vietnam. His current research interests include information security, communication protocols, communication systems, QoS mechanisms, and control systems..

Leu Duc Tan received his PhD in cryptographic techniques in 1992. His research interests include Analysis and Design of Security Protocols, and Cryptographic Theory.