Faster ECC over $\mathbb{F}_{2^{571}}$ (feat. PMULL)

Hwajeong Seo[1], Zhe Liu[2], Zhi Hu[3], Lu Zhou[4], Yasuyuki Nogami[5], Jongseok Choi[6], Taehwan Park[6], and Howon Kim[6]⋆

[1] Institute for Infocomm Research (I2R), Singapore
hwajeong84@gmail.com

[2] University of Waterloo, Institute for Quantum Computing and Department of Combinatorics and Optimization, Canada
z446liu@uwaterloo.ca

[3] Central South University, School of Mathematics and Statistics, China
huzhi_math@csu.edu.cn

[4] University of Luxembourg, Security and Trust (SnT), 6, rue R. Coudenhove-Kalergi, L–1359 Luxembourg-Kirchberg, Luxembourg
lu.zhou@uni.lu

[5] Okayama University, Graduate School of Natural Science and Technology, 3-1-1, Tsushima-naka, Kita, Okayama, 700-8530, Japan
yasuyuki.nogami@okayama-u.ac.jp

[6] Pusan National University, School of Computer Science and Engineering, San-30, Jangjeon-Dong, Geumjeong-Gu, Busan 609–735, Republic of Korea
{jschoi85,pth5804,howonkim}@pusan.ac.kr

Abstract. In this paper, we show efficient elliptic curve cryptography implementations for B-571 over ARMv8. We improve the previous binary field multiplication with finely aligned multiplication and incomplete reduction techniques by taking advantages of advanced 64-bit polynomial multiplication (PMULL) supported by ARMv8. This approach shows performance enhancements by a factor of 1.34 times than previous binary field implementations. For the point addition and doubling, the special types of multiplication, squaring and addition operations are combined together and optimized, where one reduction operation is optimized in each case. The scalar multiplication is implemented in constant-time Montgomery ladder algorithm, which is secure against timing attacks. Finally the proposed implementations achieved 759,630/331,944 clock cycles for random/fixed scalar multiplications for B-571 over ARMv8, respectively.

Keywords: ARMv8, Elliptic Curve Cryptography, Binary Field Multiplication

⋆ Corresponding Author

1 Introduction

Elliptic Curve Cryptography (ECC) is the most popular Public Key Cryptography (PKC) in modern computers. However due to its high complexities, the computations become performance bottleneck in the applications. Particularly, the binary field multiplication is regarded as the most expensive operation so many researchers have studied the high-speed implementation of binary field multiplication in order to improve the availability of applications. The classical binary field multiplication performs the bitwise exclusive-or operation with the operands and the intermediate results when the target bit of operand is set to one [14, 11, 12]. The alternative approach takes advantages of the pre-computed Look-Up Table (LUT). The method constructs the part of results in advance and then the logical operations are replaced into the simple memory access operations [7, 10]. Recently, the modern embedded processors support the advanced built-in polynomial multiplication. ARMv7 architecture supports `VMULL.P8` operation which computes eight 8-bit wise polynomial multiplications with single instruction and then outputs eight 16-bit results to the 128-bit NEON register. In [2], Câmara et al. shows that the efficient 64-bit polynomial multiplication with the `VMULL.P8` instruction. Since the `VMULL.P8` instruction only provides the outputs in vectorized formats, the author presents nice approaches to align the vectorized into sequential results. After then multiple levels of Karatsuba multiplications are applied to various binary field multiplications ranging from $\mathbb{F}_{2^{251}}$, $\mathbb{F}_{2^{283}}$ to $\mathbb{F}_{2^{571}}$. The advanced ARMv8 architecture supports `PMULL` instruction which computes the 64-bit wise polynomial multiplication. In CT-RSA'15, Gouvêa and López presented compact implementations of GCM based Authenticated Encryption (AE) with the built-in AES encryption and `PMULL` instruction [3]. Since the 128-bit polynomial multiplication only needs 4 times of `PMULL` instructions, the basic multiplication shows better performance than asymptotically faster Karatsuba multiplication. After then the authors evaluate the built-in AES encryption, which improves the performance of GCM by about 11 times than that of ARMv7. In [13], authors evaluated the `PMULL` based binary field multiplication techniques ranging from 192-bit to 576-bit for ECC. From 256-bit polynomial multiplication, Karatsuba techniques show performance enhancements than traditional approaches. However, the paper does not explore the full implementations of ECC and we found a room to improve the performance further from the work.

In this paper, we present efficient implementation techniques for B-571 on ARMv8. We improve the previous binary field multiplication by introducing finely aligned multiplication and incomplete reduction technique. The proposed technique improves the performance by a factor of 1.34 times than previous Seo et al.'s implementations. For the point addition and doubling, we perform the combined reduction on special types of binary field multiplication, squaring and addition operations. The scalar multiplication is implemented in Montgomery ladder algorithm, which ensures constant timing and security against timing attacks. Finally, we set the speed record for B-571 on ARMv8, which performs the unknown/fixed scalar multiplications within 759,630/331,944 clock cycles, respectively.

The remainder of this paper is organized as follows. In Section 2, we recap the B-571 curve, target ARM processor and previous polynomial multiplication on ARMv8. In Section 3, we propose the efficient ECC implementations on ARMv8. In Section 4, we evaluate the performance of proposed methods. Finally, Section 5 concludes the paper.

2 Related Works

2.1 Elliptic curve over $\mathbb{F}_{2^{571}}$

The 571-bit elliptic curve standardized in [1] and the finite field $\mathbb{F}_{2^m}$ is defined by:

$$f(x) = x^{571} + x^{10} + x^5 + x^2 + 1$$

The curve $E : y^2 = xy = x^3 + ax^2 + b$ over $\mathbb{F}_{2^m}$ is defined by:

```
a = 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000
                               00000000 00000000 00000000 00000001
```

```
b = 02F40E7E 2221F295 DE297117 B7F3D62F 5C6A97FF CB8CEFF1 CD6BA8CE
    4A9A18AD 84FFABBD 8EFA5933 2BE7AD67 56A66E29 4AFD185A 78FF12AA
                               520E4DE7 39BACA0C 7FFEFF7F 2955727A
```

and group order is defined by:

```
n = 03FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
    FFFFFFFF FFFFFFFF E661CE18 FF559873 08059B18 6823851E C7DD9CA1
                               161DE93D 5174D66E 8382E9BB 2FE84E47
```

2.2 ARM Processor

Advanced RISC Machine (ARM) is an instruction set architecture (ISA) for high-performance embedded applications. ARM architecture supports low-power consumptions and high code density. The most advanced ARMv8 processor supports both 32-bit (AArch32) and 64-bit (AArch64) architectures. Particularly the processor supports a single-instruction multiple-data (SIMD) instruction sets with

Algorithm 1 192-bit Polynomial Multiplication in Program Codes

```
Require: 192-bit operands $A[2 \sim 0]$ (v0, v1) and $B[2 \sim 0]$ (v5, v6).
Ensure: 384-bit result $C[5 \sim 0] \leftarrow A[2 \sim 0] \times B[2 \sim 0]$ (v10, v11, v12).
 1: pmull v10.1q, v0.1d, v5.1d                 {$A[0] \times B[0]$}
 2: pmull v11.1q, v0.1d, v6.1d                 {$A[0] \times B[2]$}
 3: pmull2 v28.1q, v0.2d, v5.2d                {$A[1] \times B[1]$}
 4: eor.16b v11, v11, v28
 5: pmull v28.1q, v1.1d, v5.1d                 {$A[2] \times B[0]$}
 6: eor.16b v11, v11, v28
 7: pmull v12.1q, v1.1d, v6.1d                 {$A[2] \times B[2]$}
 8: ext.16b v30, v0, v0, #8
 9: pmull2 v29.1q, v30.2d, v5.2d               {$A[0] \times B[1]$}
10: pmull v28.1q, v30.1d, v5.1d                {$A[1] \times B[0]$}
11: eor.16b v29, v29, v28
12: pmull v30.1q, v30.1d, v6.1d                {$A[1] \times B[2]$}
13: ext.16b v28, v1, v1, #8
14: pmull2 v28.1q, v28.2d, v5.2d               {$A[2] \times B[1]$}
15: eor.16b v30, v30, v28
16: ext.16b v28, v31, v29, #8
17: ext.16b v29, v29, v30, #8
18: ext.16b v30, v30, v31, #8
19: eor.16b v10, v10, v28
20: eor.16b v11, v11, v29
21: eor.16b v12, v12, v30
```

NEON engine. The processor has 32 64-bit registers (`X0-X31`) and 32 128-bit NEON registers (`V0-V31`). Particularly, 64-bit wise polynomial multiplication instructions (`PMULL` and `PMULL2`) are available. The `PMULL` instruction uses the lower 64-bit part in 128-bit register for the input, while the `PMULL2` instruction uses the higher 64-bit part in 128-bit register for the input [3].

2.3 Polynomial Multiplication on ARMv8

In [13], authors evaluated the `PMULL` instructions for the various polynomial multiplications ranging from 192-bit to 576-bit. Particularly, the authors perform the three terms of Karatsuba multiplication for 576-bit case, which reduces the number of 192-bit wise multiplication from 9 to 6 [8, 15, 6]. The author claims that basic approach for 192-bit case is more efficient than Karatsuba multiplication on the ARMv8 architecture, since additional number of addition operations are larger than optimized multiplication operations. The detailed program codes are drawn in Algorithm 1. The approach requires the 9 64-bit wise polynomial multiplications. The partial products ($A[0] \times B[1]$, $A[1] \times B[0]$, $A[1] \times B[2]$, $A[2] \times B[1]$) are computed and shifted by 64-bit. The shifted results are accumulated to the intermediate results for partial products ($A[0] \times B[0]$, $A[0] \times B[2]$, $A[1] \times B[1]$, $A[2] \times B[0]$, $A[2] \times B[2]$).

3 Proposed Method

3.1 Optimization for Finite Field Operation

The polynomial addition/subtraction can be performed with bit-wise exclusive-or instructions on both operands. For the 576-bit case, each operand is loaded to the 5 128-bit NEON registers ($5 = \lceil 4.5 \rceil = \lceil \frac{576}{128} \rceil$) and 5 times of bit-wise excluisve-or operations are performed.

The binary field multiplication is the most expensive operation in the finite field operations. For 576-bit case, Seo et al. proposed the three-term of Karatsuba multiplication [13]. Each term performs the classical 192-bit wise polynomial multiplication (See Algorithm 1). The 192-bit multiplication always outputs the 384-bit results in 3 consecutive 128-bit registers. However, this alignment style requires additional 64-bit wise shift operations to get aligned intermediate results in three steps for 576-bit multiplication (See Step 4, 8, 10 in Algorithm 2). In order to hide these latencies, we used both previous and shifted 192-bit polynomial multiplication. In Algorithm 3, shifted version of multiplication is described. Unlike previous approach described in Algorithm 1, the partial products ($A[0] \times B[0]$, $A[0] \times B[2]$, $A[1] \times B[1]$, $A[2] \times B[0]$, $A[2] \times B[2]$) are shifted by 64-bit. The shifted results are accumulated to the intermediate results for partial products ($A[0] \times B[1]$, $A[1] \times B[0]$, $A[1] \times B[2]$, $A[2] \times B[1]$). The 64-bit shifted results are stored into 4 consecutive NEON registers (`v16, v17, v18, v19`) where the least significant 64-bit of `v16` and most significant 64-bit of `v19` are set to zero. The detailed 576-bit multiplication is described in Algorithm 2. The 576-bit polynomial multiplication requires 6 192-bit polynomial multiplications in Step 3, 4, 5, 8, 9 and 10. The results are required to be shifted by 192, 384 or 576-bit to the left before intermediate result are accumulated. The 384-bit shift case does not require additional shift operations on the 128-bit register. However, two cases (192 and 576-bit) requires 64-bit wise shift to the left to align the results (Step 4, 8, 10). In this case, we used the shifted 192-bit polynomial multiplication described in Algorithm 3. For the other three cases (Step 3, 5, 9), we used the previous approach described in Algorithm 1. By using shifted approach, we can avoid three times of 64-bit wise shift operations in each multiplication. In instruction set level, 12 times of extraction instructions are optimized.

The polynomial squaring is a linear operation, since the result is obtained by inserting a 0 bit between consecutive bits of operand. By using the 64-bit polynomial multiplication (`PMULL`) instruction, we can compute the 64-bit wise squaring with single `PMULL` instruction. In Algorithm 4, the 576-bit wise squaring operation is drawn. The 576-bit operand requires 9 ($\frac{576}{64}$) times of `PMULL` instructions.

The m-bit polynomial multiplication/squaring operations produce the values of degree at most $2m$-bit, which must be reduced by modulo. When the modulo is smaller than operand size (64-bit) of `PMULL` instruction, we can perform the multiplication on higher parts ($> m$) by modulo. The modulo of binary field $\mathbb{F}_{2^{571}}$ is defined by $f(x) = x^{571} + x^{10} + x^5 + x^2 + 1$, which is only 11-bit modulo

Algorithm 2 Aligned Polynomial Multiplication for 576-bit

```
Require: 576-bit operands $A[8 \sim 0]$ and $B[8 \sim 0]$.
Ensure: 1152-bit result $C[17 \sim 0] \leftarrow A[8 \sim 0] \times B[8 \sim 0]$.
 1: $A \leftarrow \{A_H, A_M, A_L\} \leftarrow \{(A[8], A[7], A[6]), (A[5], A[4], A[3]), (A[2], A[1], A[0])\}$
 2: $B \leftarrow \{B_H, B_M, B_L\} \leftarrow \{(B[8], B[7], B[6]), (B[5], B[4], B[3]), (B[2], B[1], B[0])\}$
 3: $C_H \leftarrow (A_H \times_{192} B_H) \lll 384$                          { Algorithm 1}
 4: $C_M \leftarrow (A_M \times_{192} B_M) \lll 192$                          { Algorithm 3}
 5: $C_L \leftarrow A_L \times_{192} B_L$                                     { Algorithm 1}
 6: $T \leftarrow C_H \oplus C_M \oplus C_L$
 7: $C \leftarrow T \oplus (T \lll 192) \oplus (T \lll 384)$
 8: $C_H \leftarrow ((A_H \oplus A_M) \times_{192} (B_H \oplus B_M)) \lll 576$    { Algorithm 3}
 9: $C_M \leftarrow ((A_H \oplus A_L) \times_{192} (B_H \oplus B_L)) \lll 384$    { Algorithm 1}
10: $C_L \leftarrow ((A_M \oplus A_L) \times_{192} (B_M \oplus B_L)) \lll 192$    { Algorithm 3}
11: $C \leftarrow C_H \oplus C_M \oplus C_L$
```

Algorithm 3 (Shifted) 192-bit Polynomial Multiplication in Program Codes

```
Require: 192-bit operands $A[2 \sim 0]$ (v1, v2) and $B[2 \sim 0]$ (v6, v7).
Ensure: 384-bit result $C[5 \sim 0] \leftarrow A[2 \sim 0] \times B[2 \sim 0]$ (v16, v17, v18, v19).
 1: pmull v17.1q, v1.1d, v6.1d            { $A[0] \times B[0]$}
 2: pmull v18.1q, v1.1d, v7.1d            { $A[0] \times B[2]$}
 3: pmull2 v28.1q, v1.2d, v6.2d           { $A[1] \times B[1]$}
 4: eor.16b v18, v18, v28
 5: pmull v28.1q, v2.1d, v6.1d            { $A[2] \times B[0]$}
 6: eor.16b v18, v18, v28
 7: pmull v19.1q, v2.1d, v7.1d            { $A[2] \times B[2]$}
 8: ext.16b v16, v31, v17, #8
 9: ext.16b v17, v17, v18, #8
10: ext.16b v18, v18, v19, #8
11: ext.16b v19, v19, v31, #8
12: ext.16b v30, v1, v1, #8
13: pmull2 v29.1q, v30.2d, v6.2d          { $A[0] \times B[1]$}
14: pmull v28.1q, v30.1d, v6.1d           { $A[1] \times B[0]$}
15: eor.16b v29, v29, v28
16: pmull v30.1q, v30.1d, v7.1d           { $A[1] \times B[2]$}
17: ext.16b v28, v2, v2, #8
18: pmull2 v28.1q, v28.2d, v6.2d          { $A[2] \times B[1]$}
19: eor.16b v30, v30, v28
20: eor.16b v17, v17, v29
21: eor.16b v18, v18, v30
```

Algorithm 4 576-bit Polynomial Squaring

```
Require: 576-bit Operand $A[8 \sim 0]$.
Ensure: 1152-bit Result $C[17 \sim 0] \leftarrow A[8 \sim 0] \times A[8 \sim 0]$.
1: for i = 0 to 8 by 1 do
2:     $\{C[2 \times i + 1] || C[2 \times i]\} \leftarrow A[i] \times A[i]$
3: end for
```

Algorithm 5 Fast Reduction over $\mathbb{F}_{2^{571}}$

```
Require: 576-bit (complete) or 1152-bit (incomplete) operands $A$, complete reduction.
Ensure: 571-bit (complete) or 576-bit (incomplete) result $C$.
 1: if complete reduction then
 2:     $r \leftarrow$0x425
 3:     $A_L \leftarrow A$ mod $2^{571}$
 4:     $A_H \leftarrow A$ div $2^{571}$
 5:     $T \leftarrow A_H \times r$
 6:     $C \leftarrow A_L \oplus T$
 7: else
 8:     $r \leftarrow$0x84A0
 9:     $A_L \leftarrow A$ mod $2^{576}$
10:     $A_H \leftarrow A$ div $2^{576}$
11:     $T \leftarrow A_H \times r$
12:     $T \leftarrow A_L \oplus T$
13:     $T_L \leftarrow T$ mod $2^{576}$
14:     $T_H \leftarrow T$ div $2^{576}$
15:     $T \leftarrow T_H \times r$
16:     $C \leftarrow T_L \oplus T$
17: end if
```

so we can use `PMULL` instruction for computations. However, 571-bit modulo is not efficient over the 64-bit machine since this requires 5-bit wise shift operations to align the results. Alternatively, we choose the 64-bit machine friendly modulo ($f(x) = x^{576} + x^{15} + x^{10} + x^7 + x^5$) and incomplete reduction. This approach avoids the number of 5-bit wise shift operations and complete results are also obtained by performing the complete reduction before outputting the results. The detailed reduction process is available in Algorithm 5. If the complete reduction is selected, the modulo (r) is set to `0x425` representing the values ($x^{10} + x^5 + x^2 + 1$). In Step 3, the part of A which is lower than 571-bit is extracted to A_L. In Step 4, the part of A which is higher than 571-bit is extracted to A_H. In Step 5, the higher part (A_H) is multiplied by modulus (r). In Step 6, the results are added to the lower part (A_L). In case of incomplete reduction, the modulus (r) is set to `0x84A0` representing the values ($x^{15} + x^{10} + x^7 + x^5$). In Step 9, the part of A which is lower than 576-bit is extracted to A_L. In Step 10, the part of A which is higher than 576-bit is extracted to A_H. In Step 11, the higher part (A_H) is multiplied by modulus (r). In Step 12, the lower part (A_L) are added to the intermediate results T. In Step 13, the part of T which is lower than 576-bit is extracted to T_L. In Step 14, the part of T which is higher than 576-bit is extracted to T_H. In Step 15, the higher part (T_H) is multiplied by modulus (r). In Step 16, the lower part (T_L) are added to the intermediate results T.

For fast and secure inversion operation, we used the Itoh-Tsujii algorithm [4], which is an optimization of inversion through Fermat's little theorem ($f(x)^{-1} = f(x)^{2^m-2}$), ensuring the constant time computations. The algorithm uses a repeated field squaring and multiplication operations for $f(x)^{2^k}$, which follows a chains of multiplication and squaring sequences ($f_1 \rightarrow f_2 \rightarrow f_4 \rightarrow f_8 \rightarrow f_{16} \rightarrow f_{17} \rightarrow f_{34} \rightarrow f_{35} \rightarrow f_{70} \rightarrow f_{71} \rightarrow f_{142} \rightarrow f_{284} \rightarrow f_{285} \rightarrow f_{570}$). The inversion algorithm requires 13 multiplication and 570 squaring operations.

3.2 Optimization for Scalar Multiplication

In order to perform the scalar multiplication, the point addition and doubling operations are required, which consist of a number of finite field operations. Depending on specific coordinates, the number of finite field operations are varied each other. The point addition in López-Dahab/affine coordinates requires 8 multiplication (`M`), 5 squaring (`S`) and 1 a-multiplication (`a-M`). Alternative point addition in López-Dahab coordinates requires 13`M` and 5`S`. For the point doubling in López-Dahab coordinates requires 3`M`, 5`S`, 1`a-M` and 1 b-multiplication (`b-M`). Particularly, the variable (a) is set to 1 in the B-571 curve so the `a-M` operation is free. The binary field multiplication and squaring operations are performed by following the implementation techniques described in Section 3.1. A sequence of multiplication, squaring and addition operations are optimized again by combining the reduction operations . This sequence of field operations involve a type ($A \times B + C \times D$). The straight-forward implementation of type requires 2 multiplication, 2 reduction and 1 addition operations. One reduction operation can be optimized by performing the multiplication and addition operations in advance [9]. Similar a type ($A^2 + C \times D$) is also optimized from 1 squaring, 1 multiplication, 2 reduction and 1 addition operations to 1 squaring, 1 multiplication, 1 reduction and 1 addition operations. We employed the Negre and Robert techniques for the point addition in López-Dahab/affine coordinates and doubling in López-Dahab coordinates. For point addition in López-Dahab/affine coordinates described in Algorithm 6, Step 13 and 19 can be optimized through optimal ($A^2 + C \times D$) and ($A \times B + C \times D$) types. For point doubling in López-Dahab coordinates described in Algorithm 7, Step 12 can be optimized through optimal ($A \times B + C \times D$) type. We extended this technique to point addition in López-Dahab coordinates in Algorithm 8. The Step 17, 18 and 20 include the ($A \times B + C \times D$) type and this approach optimizes the 3 reduction operations in each point addition operation.

The scalar multiplication is implemented in Montgomery ladder algorithm [5]. This algorithm always performs the point addition and doubling operations in each bit and our implementations of finite field arithmetic are also regular fashion, which ensure constant-time computation and security against Simple Power Analysis (SPA). For unknown point, we used point addition/doubling in López-Dahab coordinates with window methods and for fixed point we used point addition in López-Dahab/affine coordinates and doubling in López-Dahab coordinates with window methods.

4 Evaluation

We used Xcode (ver 6.3.2) as a development IDE and programmed over iPad Mini2 (iOS 8.4). The iPad Mini2 equipped Apple A7 with 64-bit architecture operated in the frequency of 1.3GHz. The program is written in C and assembly codes and complied with `-Ofast` optimization level. The timing are acquired through the clock cycles of real device.

Algorithm 6 Optimization for Point Addition in López-Dahab/affine coordinates [9]

```
Require: Point $P1$ $(X1, Y1, Z1)$ in López-Dahab coordinates and $P2$ $(X2, Y2, 1)$ in affine coordinates
Ensure: Point $P3$ $(X3, Y3, Z3)$ in López-Dahab coordinates
 1: $t0 \leftarrow Z1^2$
 2: $t1 \leftarrow Y2 \times t0$
 3: $k0 \leftarrow Y1 + t1$
 4: $t2 \leftarrow X2 \times Z1$
 5: $k1 \leftarrow X1 + t2$
 6: $k2 \leftarrow k1 \times Z1$
 7: $Z3 \leftarrow k2^2$
 8: $k4 \leftarrow X2 \times Z3$
 9: $t3 \leftarrow k1^2$
10: $t4 \leftarrow a \times k2$
11: $t5 \leftarrow k0 + t3$
12: $t6 \leftarrow t5 + t4$
13: $X3 \leftarrow k0^2 + k2 \times t6$    $\{A^2 + C \times D\}$
14: $t7 \leftarrow k0 \times k2$
15: $t8 \leftarrow k4 + X3$
16: $t9 \leftarrow t7 + Z3$
17: $t10 \leftarrow Y2 + X2$
18: $t11 \leftarrow Z3^2$
19: $Y3 \leftarrow t10 \times t11 + t8 \times t9$    $\{A \times B + C \times D\}$
```

Algorithm 7 Optimization for Point Doubling in López-Dahab coordinates [9]

```
Require: Point P1 (X1, Y1, Z1) in López-Dahab coordinates
Ensure: Point P3 (X3, Y3, Z3) in López-Dahab coordinates
 1: $k0 \leftarrow Z1^2$
 2: $t0 \leftarrow k0^2$
 3: $k1 \leftarrow b \times t0$
 4: $k2 \leftarrow X1^2$
 5: $Z3 \leftarrow A \times k2$
 6: $t1 \leftarrow k2^2$
 7: $X3 \leftarrow t1 + k1$
 8: $t2 \leftarrow Y1^2$
 9: $t3 \leftarrow a \times Z3$
10: $t4 \leftarrow t2 + t3$
11: $t5 \leftarrow t4 + k1$
12: $Y3 \leftarrow t5 \times X3 + Z3 \times k1$    $\{A \times B + C \times D\}$
```

Algorithm 8 Optimization for Point Addition in López-Dahab coordinates

```
Require: Point $P1$ $(X1, Y1, Z1)$ and $P2$ $(X2, Y2, Z2)$ in López-Dahab coordinates
Ensure: Point $P3$ $(X3, Y3, Z3)$ in López-Dahab coordinates
 1: $k0 \leftarrow X1 \times Z2$
 2: $k1 \leftarrow X2 \times Z1$
 3: $k2 \leftarrow k0^2$
 4: $k3 \leftarrow k1^2$
 5: $k4 \leftarrow k0 + k1$
 6: $k5 \leftarrow k2 + k3$
 7: $t0 \leftarrow Z2^2$
 8: $k6 \leftarrow Y1 \times t0$
 9: $t1 \leftarrow Z1^2$
10: $k7 \leftarrow Y2 \times t1$
11: $k8 \leftarrow k6 + k7$
12: $k9 \leftarrow k8 \times k4$
13: $t2 \leftarrow Z1 \times Z2$
14: $Z3 \leftarrow k5 \times t2$
15: $t3 \leftarrow k7 \times k3$
16: $t4 \leftarrow k2 \times k6$
17: $X3 \leftarrow k1 \times t4 + k0 \times t3 \{A \times B + C \times D\}$
18: $t5 \leftarrow k5 \times k6 + k0 \times k9 \{A \times B + C \times D\}$
19: $t6 \leftarrow k9 + Z3$
20: $Y3 \leftarrow t6 \times X3 + t5 \times k5 \{A \times B + C \times D\}$
```

Table 1: Comparison results of binary field multiplication for B-571 curve

Algorithm	Clock cycle
Seo et al. [13]	132
Proposed Method	99

Table 2: Performance evaluations of B-571 curve, where w is window size

Operation	Clock cycle
Binary Field Operation	
Multiplication	99
Squaring	24
Inversion	31,232
Group Operation	
Point addition (LD/affine)	1,107
Point addition (LD)	1,537
Point doubling (LD)	609
Scalar Multiplication	
Unknown point ($w = 4$)	759,630
Fixed point ($w = 4$)	331,944

In Table 1, the comparison results of binary field multiplication over B-571 curve are drawn. We only compared results with Seo et al. [13] since SUPERCOP benchmark tool does not support the iOS operating system which is required for our experiments and the work by Gouvêa and J. López is only provide the GCM operations [3]. The Seo et al. achieved the high performance with three-term of Karatsuba multiplication for 576-bit polynomial multiplication and fast reduction techniques. In our implementation, we further improved performance by a factor of 1.34 times with the finely aligned multiplications and incomplete reduction techniques.

In Table 2, we listed the whole results of ECC implementations. Unfortunately, there is no paper about ECC implementations on ARMv8. We only provide our results. The squaring operation is linear computations, which requires small number of clock cycles. The inversion operation is implemented in Fermat's little theorem, which requires 570 squaring and 13 multiplication operations. For group operations, three different point operations are evaluated. The doubling in López-Dahab coordinates shows the lowest clock cycles. The point addition in López-Dahab coordinates shows the highest clock cycles. Finally, the scalar multiplication is efficiently implemented with window methods. In the fixed point, points can be pre-computed and the number of doubling operations are optimized. In this paper, we explore the medium window size but this can be easily extended to the long window size by sacrificing the RAM storages.

5 Conclusion

In this paper, we show efficient finite field and group operations for B-571 ECC implementations over ARMv8. We optimized the binary field arithmetics by introducing the several optimization techniques. The group operations are also improved by reducing the number of reduction operations in point addition and doubling operations. Finally, we achieved the high speed implementation of B-571 implementation over ARMv8.

6 Conflict of Interests

The author(s) declare(s) that there is no conflict of interest regarding the publication of this paper.

References

1. Recommended elliptic curve domain parameters. *Standards for Efficient Cryptography Group, Certicom Corp*, 2000.
2. D. Câmara, C. P. Gouvêa, J. López, and R. Dahab. Fast software polynomial multiplication on ARM processors using the NEON engine. In *International Conference on Availability, Reliability, and Security*, pages 137–154. Springer, 2013.
3. C. P. Gouvêa and J. López. Implementing GCM on ARMv8. In *Cryptographers Track at the RSA Conference*, pages 167–180. Springer, 2015.
4. T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in GF(2m) using normal bases. *Information and computation*, 78(3):171–177, 1988.
5. M. Joye and S.-M. Yen. The Montgomery powering ladder. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 291–302. Springer, 2002.
6. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.
7. J. López and R. Dahab. High-speed software multiplication in GF(2^m). In *International Conference on Cryptology in India*, pages 203–212. Springer, 2000.
8. P. L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Transactions on Computers*, 54(3):362–369, 2005.
9. C. Negre and J.-M. Robert. Impact of optimized field operations AB, AC and AB+CD in scalar multiplication over binary elliptic curve. In *International Conference on Cryptology in Africa*, pages 279–296. Springer, 2013.
10. L. B. Oliveira, D. F. Aranha, C. P. Gouvêa, M. Scott, D. F. Câmara, J. López, and R. Dahab. TinyPBC: Pairings for authenticated identity-based non-interactive key distribution in sensor networks. *Computer Communications*, 34(3):485–493, 2011.
11. H. Seo, Y. Lee, H. Kim, T. Park, and H. Kim. Binary and prime field multiplication for public key cryptography on embedded microprocessors. *Security and Communication Networks*, 7(4):774–787, 2014.
12. H. Seo, Z. Liu, J. Choi, and H. Kim. Karatsuba–block-comb technique for elliptic curve cryptography over binary fields. *Security and Communication Networks*, 8(17):3121–3130, 2015.
13. H. Seo, Z. Liu, Y. Nogami, J. Choi, and H. Kim. Binary field multiplication on ARMv8. *Security and Communication Networks*, 2016.

14. M. Shirase, Y. Miyazaki, T. Takagi, and H. Dong-Guk. Efficient implementation of pairing-based cryptography on a sensor node. *IEICE transactions on information and systems*, 92(5):909–917, 2009.
15. A. Weimerskirch and C. Paar. Generalizations of the Karatsuba algorithm for efficient implementations. *IACR Cryptology ePrint Archive*, 2006:224, 2006.