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

Hwajeong Seo[1], Zhe Liu[2], Yasuyuki Nogami[3], Jongseok Choi[1], and Howon Kim[1⋆]

[1] Pusan National University,
School of Computer Science and Engineering,
San-30, Jangjeon-Dong, Geumjeong-Gu, Busan 609–735, Republic of Korea
{hwajeong,jschoi85,howonkim}@pusan.ac.kr
[2] University of Luxembourg,
Laboratory of Algorithmics, Cryptology and Security (LACS),
6, rue R. Coudenhove-Kalergi, L–1359 Luxembourg-Kirchberg, Luxembourg
{zhe.liu}@uni.lu
[3] 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

Abstract. In this paper, we show efficient implementations of K-571 over ARMv8. We exploit an advanced 64-bit polynomial multiplication (`PMULL`) supported by ARMv8 for high speed multiplication and squaring operations. Particularly, multiplication is conducted with three terms of asymptotically faster Karatsuba multiplication. Inversion is constructed by using constant time Fermat-based inversion method. For high speed scalar multiplication, 4TNAF method is exploited which takes an advantage of simple doubling method. Finally, our method conducts ECDH over K-571 within 783,705 clock cycles. Our proposed method on ARMv8 improves the performance by a factor of 4.6 times than previous techniques on ARMv7.

Keywords: Polynomial Multiplication, Binary Field Multiplication, ARMv8, Elliptic Curve Cryptography, Karatsuba Multiplication, Koblitz Curve

1 Introduction

Since binary field multiplication is an important component of elliptic curve cryptography and authenticated encryption, many researches have studied the high speed implementation of binary field multiplication in software engineering. The typical binary field multiplication over embedded processor may compute the results with bitwise-xor and logical shift operations. The other more clever approach exploits the look-up table by calculating the part of results in advance [9, 10, 13, 11, 12]. Recently, many modern embedded processors adopt the

⋆ Corresponding Author

advanced built-in binary field multiplication. ARMv7 supports `VMULL.P8` operation which can compute eight 8-bit wise polynomial multiplications with single instruction. In [3], author shows that efficient implementation techniques to construct the 64-bit binary field multiplication with the `VMULL.P8` operation. After then multiple levels of Karatsuba multiplication is applied to several binary field multiplications including $\mathbb{F}_{2^{251}}$, $\mathbb{F}_{2^{283}}$ and $\mathbb{F}_{2^{571}}$. The most recent processor, ARMv8, supports `PMULL` operation which can compute 64-bit wise polynomial multiplication with single instruction. In [5], author shows that compact implementation of GCM based authenticated encryption with the `PMULL` operation. Since the 64-bit multiplication is quite fast enough for 128-bit multiplication, they avoid Karatsuba multiplication. The implementation of GCM achieved 11 times faster results than ARMv7. However, the paper does not show binary field multiplication for long length operands. The long operands are required to compute ECC based cryptography. In this paper, we present efficient implementations of K-571 curve on ARMv8. We exploit the new `PMULL` operation and applied to various binary field arithmetics. Finally, our compact implementation achieved 4.6 times faster than ARMv7 implementations.

The remainder of this paper is organized as follows. In Section 2, we recap the K-571 Koblitz curve, target ARM processor and Karatsuba method. In Section 3, we propose the efficient binary field multiplication. In Section 4, we evaluate the performance of proposed methods in terms of clock cycles. Finally, Section 5 concludes the paper.

2 Related Works

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

The 571-bit Koblitz elliptic curve namely K-571 standardized in [4] 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:

$$a = \texttt{00000000 00000000 00000000 00000000 00000000 00000000}$$
$$\texttt{00000000 00000000 00000000 00000000 00000000 00000000 00000000}$$
$$\texttt{00000000 00000000 00000000 00000000 00000000}$$

$$b = \texttt{00000000 00000000 00000000 00000000 00000000 00000000}$$
$$\texttt{00000000 00000000 00000000 00000000 00000000 00000000 00000000}$$
$$\texttt{00000000 00000000 00000000 00000000 00000001}$$

and group order is defined by:

$$n = \texttt{02000000 00000000 00000000 00000000 00000000 00000000}$$
$$\texttt{00000000 00000000 00000000 131850E1 F19A63E4 B391A8DB 917F4138}$$
$$\texttt{B630D84B E5D63938 1E91DEB4 5CFE778F 637C1001}$$

For a point $P_2 = (X_2, Y_2, 1)$ which is affine point and not equal to P_1, let $P_3 = (X_3, Y_3, Z_3) = P_1 + P_2$. Then P_3 is computed with 8 multiplication, 5 squaring, one a2 and 9 addition operations as follows [1]:

$$A \leftarrow Y_1 + Y_2 \cdot Z_1^2,\ B \leftarrow X_1 + X_2 \cdot Z_1,\ C \leftarrow B \cdot Z_1,$$
$$Z_3 \leftarrow C^2,\ D \leftarrow X_2 \cdot Z_3,\ X_3 \leftarrow A^2 + C \cdot (A + B^2 + a2 \cdot C),$$
$$Y_3 \leftarrow (D + X_3) \cdot (A \cdot C \cdot Z_3) + (Y_2 + X_2) \cdot Z_3^2$$

The K-571 curve satisfies the *Frobenius map* $\tau : E(\mathbb{F}_2^m) \rightarrow E(\mathbb{F}_2^m)$ is defined by:

$$\tau(\infty) = \infty,\ \tau(x, y) = (x^2, y^2)$$

The *Frobenius map* can be efficiently computed since squaring in $\mathbb{F}_2^m$ is relatively inexpensive.

2.2 ARM Processor

ARM processor is a well known family of RISC processor architectures introduced in 1985 [14]. The most recent version, ARMv8, supports both 32-bit and 64-bit processing. The 32-bit ARMv8 architecture is known as AArch32, while the 64-bit is known as AArch64. An ARMv8 processor can support both, allowing the execution of 32-bit and 64-bit applications. ARM processors support a single-instruction multiple-data (SIMD) module called the NEON engine. AArch32 features sixteen 32-bit registers (`R0-R15`) and sixteen 128-bit NEON registers (`Q0-Q15`). The NEON registers can also be viewed as pairs of 64-bit registers (`D0-D32`). For example, `D0` and `D1` are the lower and higher parts of `Q0`, respectively. AArch64 features thirty two 64-bit registers (`X0-X31`) and thirty two 128-bit NEON registers (`V0-V31`). The NEON registers can no longer be viewed as pairs of 64-bit registers. From ARMv8, two polynomial dedicated instructions, `PMULL` and `PMULL2`, are available. Both of which carry out a single 64-bit multiplication. In both cases, the inputs are 128-bit registers. Their difference is that in `PMULL` the lower 64-bit parts of the inputs are used as operands, while in `PMULL2` the higher 64-bit parts are used [5].

2.3 Karatsuba Algorithm

Of finite field arithmetics, multiplication operation consumes massive body of overheads. In order to accelerate the performance, we can exploit the Karatsuba

method. The basic idea of Karatsuba multiplication is to split a multiplication of two s words operands into three multiplications of size $\frac{s}{2}$, which is possible at the expense of some additions [8]. Taking the multiplication of s words operands A and B as an example, we represent the operands as $A = A_H \cdot 2^{\frac{s}{2}} + A_L$ and $B = B_H \cdot 2^{\frac{s}{2}} + B_L$. The multiplication $P = A \cdot B$ can be computed according to the Equation 1.

$$A_H \cdot B_H \cdot 2^s + [(A_H + A_L)(B_H + B_L) - A_H \cdot B_H - A_L \cdot B_L] \cdot 2^{\frac{s}{2}} + A_L \cdot B_L \quad (1)$$

Karatsuba method roughly executes $\frac{3s^2}{4}$ `mul` instructions to multiply two s-word operands [6]. Recently, the refined Karatsuba's algorithm from a Crypto 2009 paper by Bernstein [2] makes efficient use of the available registers to keep the low overheads from `load` and `store` instructions.

3 Proposed Method

3.1 Polynomial Addition

The polynomial addition is executed with bit-wise exclusive-or operation. For 571-bit operand, nine times of 64-bit wise exclusive-or operations are required. The detailed algorithm and source code are available in Algorithm 1 and 7, respectively.

Algorithm 1 571-bit Polynomial Addition

Require: 571-bit Operands A and B.
Ensure: 571-bit Result $C = A \oplus B$.
1: **for** i = 0 to 8 by 1 **do**
2: $C[i] = A[i] \oplus A[i]$
3: **end for**

3.2 Polynomial Multiplication

Polynomial multiplication can be implemented in ordinary multiplication or Karatsuba method. The Karatsuba multiplication is an efficient approach when size of operand is long enough than processor's word size. The efficient Karatsuba multiplication techniques are highly relied on the number of terms where term is calculated in following equation (operand size/word size). In case of our target operand (571-bit), Karatsuba approach would be better choice due to its long operands. Our method combines the both Karatsuba algorithm and a new multiplier based on `PMULL` and we named the method as Karatsuba/NEON/`PMULL` multiplier (KNP). Since the 571-bit operands have 192-bit nine terms, straightforward Karatsuba implementation introduces the high complexity. We firstly

divide 571-bit operands into three 192-bit operands. For the 192-bit operand multiplication, ordinary multiplication method is the more efficient than that of Karatsuba algorithm. The comparison results are drawn in Table 1. Three terms of Karatsuba multiplication reduces the number of multiplication from 9 to 6. However, additional 8 and 5 times of `eor` and `ext` instructions are required. For this reason, ordinary multiplication is better choice for 192-bit polynomial multiplication. The detailed 192-bit wise polynomial multiplication is available in Algorithm 2. In Step 1 and 2, 192-bit operands (A and B) are loaded from memory. By using option 8b, we loaded operands by sequential 64-bit format to the 128-bit registers, which lefts higher 64-bit as an empty. In Step 3 $\sim$ 6, four multiplications including ($C_L \leftarrow A_{[63:0]} \cdot B_{[63:0]}$, $T_L \leftarrow A_{[63:0]} \cdot B_{[127:64]}$, $C_M \leftarrow A_{[63:0]} \cdot B_{[191:128]}$ and $Temp \leftarrow A_{[127:64]} \cdot B_{[63:0]}$) are conducted. After then results ($A_{[127:64]} \cdot B_{[63:0]}$) are added to T_L. From Step 8 to 15, remaining multiplications are conducted and then added to the intermediate results. After then the results are aligned and accumulated to the intermediate results in Step 17 $\sim$ 22. Finally, total 384-bit results are stored into memory.

Table 1. Comparison of 192-bit polynomial multiplication methods

Instructions	`pmull`	`eor`	`movi`	`ext`
Ordinary	9	7	1	3
Karatsuba	6	16	1	8

After then we conduct the 571-bit multiplication with the 192-bit wise multiplication (refer Algorithm 2). We can see the 571-bit multiplication as a three terms of multiplication where each term has 192-bit. On these three terms, we applied three terms of Karatsuba algorithm. The three terms of Karatsuba multiplication replaces two s words operands into six multiplications of size $\frac{s}{3}$, which is possible at the expense of some additions [8]. Taking the multiplication of s words operands A and B as an example, we represent the operands as $A = A_H \cdot 2^{\frac{2s}{3}} + A_M \cdot 2^{\frac{s}{3}} + A_L$ and $B = B_H \cdot 2^{\frac{2s}{3}} + B_M \cdot 2^{\frac{s}{3}} + B_L$. The multiplication $P = A \cdot B$ can be computed according to the Equation 1.

$$\begin{aligned}(A_H \cdot B_H \cdot 2^{\frac{2s}{3}} + A_M \cdot B_M \cdot 2^{\frac{s}{3}} + A_L \cdot B_L) \cdot (2^{\frac{2s}{3}} + 2^{\frac{s}{3}} + 1) + \\ (A_H + A_M) \cdot (B_H + B_M) \cdot 2^s + (A_H + A_L) \cdot (B_H + B_L) \cdot 2^{\frac{2s}{3}} + \\ (A_M + A_L) \cdot (B_M + B_L) \cdot 2^{\frac{s}{3}}\end{aligned} \tag{1}$$

The pseudo code for 571-bit polynomial multiplication is available in Algorithm 3. In Step 1 and 2, we group the nine operands $\{(A[8], A[7], A[6]), (A[5], A[4], A[3]), (A[2], A[1], A[0])\}$ and $\{(B[8], B[7], B[6]), (B[5], B[4], B[3]), (B[2], B[1], B[0])\}$ into three groups $\{A_H, A_M, A_L\}$ and $\{B_H, B_M, B_L\}$. In Step 3 $\sim$ 6, 192-bit wise partial products of $A_H \times_{192} B_H$, $A_M \times_{192} B_M$ and $A_L \times_{192} B_L$ are computed and then added to intermediate results T. In Step 7, the intermediate

Algorithm 2 192-bit Polynomial Multiplication (`mul192_p64`)

Require: 192-bit Operands A, B.
Ensure: 384-bit Result C.

1:	`ld1.8b {v0, v1, v2}, [x2]`	{Load 192-bit Operand A}
2:	`ld1.8b {v3, v4, v5}, [x1]`	{Load 192-bit Operand B}
3:	`pmull v6.1q, v0.1d, v3.1d`	$\{C_L \leftarrow A_{[63:0]} \cdot B_{[63:0]}\}$
4:	`pmull v9.1q, v0.1d, v4.1d`	$\{T_L \leftarrow A_{[63:0]} \cdot B_{[127:64]}\}$
5:	`pmull v7.1q, v0.1d, v5.1d`	$\{C_M \leftarrow A_{[63:0]} \cdot B_{[191:128]}\}$
6:	`pmull v11.1q, v1.1d, v3.1d`	$\{Temp \leftarrow A_{[127:64]} \cdot B_{[63:0]}\}$
7:	`eor.16b v9, v9, v11`	$\{T_L \leftarrow T_L \oplus Temp\}$
8:	`pmull v11.1q, v1.1d, v4.1d`	$\{Temp \leftarrow A_{[127:64]} \cdot B_{[127:64]}\}$
9:	`eor.16b v7, v7, v11`	$\{C_M \leftarrow C_M \oplus Temp\}$
10:	`pmull v10.1q, v1.1d, v5.1d`	$\{T_H \leftarrow A_{[127:64]} \cdot B_{[191:128]}\}$
11:	`pmull v11.1q, v2.1d, v3.1d`	$\{Temp \leftarrow A_{[191:128]} \cdot B_{[63:0]}\}$
12:	`eor.16b v7, v7, v11`	$\{C_M \leftarrow C_M \oplus Temp\}$
13:	`pmull v11.1q, v2.1d, v4.1d`	$\{Temp \leftarrow A_{[191:128]} \cdot B_{[127:64]}\}$
14:	`eor.16b v10, v10, v11`	$\{T_H \leftarrow T_H \oplus Temp\}$
15:	`pmull v8.1q, v2.1d, v5.1d`	$\{C_H \leftarrow A_{[191:128]} \cdot B_{[191:128]}\}$
16:	`movi.16b v11, #0`	{Clear Reg}
17:	`ext.16b v11, v11, v9 , #8`	{Align Result}
18:	`ext.16b v9, v9, v10 , #8`	{Align Result}
19:	`ext.16b v10, v10, v11, #8`	{Align Result}
20:	`eor.16b v6, v6, v11`	$\{C_{L[127:64]} \leftarrow C_{L[127:64]} \oplus T_{L[63:0]}\}$
21:	`eor.16b v7, v7, v9`	$\{C_M \leftarrow C_M \oplus \{T_{H[63:0]} \| \| T_{L[127:64]}\}\}$
22:	`eor.16b v8, v8, v10`	$\{C_{H[63:0]} \leftarrow C_{H[63:0]} \oplus T_{H[127:64]}\}$
23:	`st1.16b {v6,v7,v8}, [x0]`	{Return 384-bit Result C}

results are shifted and added to the intermediate results. In Step $8 \sim 10$, a pair of operands are added and then multiplied each other. Finally the results are added to the intermeidate results.

3.3 Polynomial Squaring

Since squaring a binary polynomial is a linear operation, this is much faster than multiplying two polynomials. The polynomial squaring is obtained by inserting a 0 bit between consecutive bits of operand. With the `PMULL` instruction, single multiplication can generate the 64-bit wise squaring at once. Finally, we can get the 571-bit squaring operation by conducting nine times of `PMULL` operation as described in Algorithm 4.

3.4 Binary Field Reduction

s word of binary field multiplication produce values of degree at most $2s - 2$, which must be reduced modulo $f(z) = z^m + r(z)$. The usual approach is to multiply the higher parts by $r(z)$ using shift and xors. For small polynomials $r(z)$ we can exploit the `PMULL` instruction to carry out 64-bit multiplication by

Algorithm 3 571-bit Polynomial Multiplication

Require: 571-bit Operands A and B.
Ensure: 1142-bit Result $C = A \cdot B$.
1: $A = \{A_H, A_M, A_L\} = \{(A[8], A[7], A[6]), (A[5], A[4], A[3]), (A[2], A[1], A[0])\}$
2: $B = \{B_H, B_M, B_L\} = \{(B[8], B[7], B[6]), (B[5], B[4], B[3]), (B[2], B[1], B[0])\}$
3: $C_H = (A_H \times_{192} B_H) \ll 384$
4: $C_M = (A_M \times_{192} B_M) \ll 192$
5: $C_L = A_L \times_{192} B_L$
6: $T = C_H \oplus C_M \oplus C_L$
7: $C = T \oplus (T \ll 192) \oplus (T \ll 384)$
8: $C_H = ((A_H \oplus A_M) \times_{192} (B_H \oplus B_M)) \ll 576$
9: $C_M = ((A_H \oplus A_L) \times_{192} (B_H \oplus B_L)) \ll 384$
10: $C_L = ((A_M \oplus A_L) \times_{192} (B_M \oplus B_L)) \ll 192$
11: $C = C_H \oplus C_M \oplus C_L$

Algorithm 4 571-bit Polynomial Squaring

Require: 571-bit Operand A.
Ensure: 1142-bit Result $C = A^2$.
1: **for** i = 0 to 8 by 1 **do**
2: $\{C[i+1]||C[i]\} = A[i] \times_{64} A[i]$
3: **end for**

$r(z)$. The modulo of binary field $\mathbb{F}_{2^{571}}$ is defined by $(r(z) = z^{10} + z^5 + z^2 + 1)$. The detailed reduction method is available in Algorithm 5. In Step 1, modulus p is set to `0x425`. In Step 2, lower part of A by 571-bit is extracted to A_L. In Step 3, higher part of A by 571-bit is extracted to A_H. In Step 4, higher part A_H is multiplied by modulus p and then added to the lower part A_L. From Step 5 to 7, one more reduction is conducted to handle the parts beyond the 571-bit.

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

Require: 1142-bit Operands A.
Ensure: 571-bit Result C.
1: $p =$`0x425`
2: $A_L = A$ `mod` 2^{571}
3: $A_H = A$ `div` 2^{571}
4: $T = A_L \oplus (A_H \cdot p)$
5: $T_L = T$ `mod` 2^{571}
6: $T_H = T$ `div` 2^{571}
7: $C = T_L \oplus (T_H \cdot p)$

3.5 Binary Field Inversion

For fast and secure against timing attack, we used the Itoh-Tsujii algorithm [7], which is an optimization of inversion through Fermat's little theorem $(a(x)^{-1} =$

$a(x)^{2^m-2}$. The algorithm uses a repeated field squaring and multiplication operations for $a(x)^{2^k}$. The multiplication and squaring are conducted with NEON KNP instruction and detailed descriptions are available in Algorithm 6. The inversion operation requires only 13 multiplication and 570 squaring operations. For implementation, we conduct the multiplication and squaring in assembly and combine the both operations in C language.

Algorithm 6 Fermat-based inversion mod $\mathbb{F}_{2^{571}}$

Require: Integer a_1 satisfying $1 \leq a_1 \leq 2^m$.
Ensure: Inverse $z = a_1^{2^m-2} = a_1^{-1}$.

1:	$a_2 \leftarrow (a_1)^{2^1} \cdot a_1$	{cost: 1S+1M}
2:	$a_4 \leftarrow (a_2)^{2^2} \cdot a_2$	{cost: 2S+1M}
3:	$a_8 \leftarrow (a_4)^{2^4} \cdot a_4$	{cost: 4S+1M}
4:	$a_{16} \leftarrow (a_8)^{2^8} \cdot a_8$	{cost: 8S+1M}
5:	$a_{17} \leftarrow (a_{16})^{2^1} \cdot a_1$	{cost: 1S+1M}
6:	$a_{34} \leftarrow (a_{17})^{2^{17}} \cdot a_{17}$	{cost: 17S+1M}
7:	$a_{35} \leftarrow (a_{34})^{2^1} \cdot a_1$	{cost: 1S+1M}
8:	$a_{70} \leftarrow (a_{35})^{2^{35}} \cdot a_{35}$	{cost: 35S+1M}
9:	$a_{71} \leftarrow (a_{70})^{2^1} \cdot a_1$	{cost: 1S+1M}
10:	$a_{142} \leftarrow (a_{71})^{2^{71}} \cdot a_{71}$	{cost: 71S+1M}
11:	$a_{284} \leftarrow (a_{142})^{2^{142}} \cdot a_{142}$	{cost: 142S+1M}
12:	$a_{285} \leftarrow (a_{284})^{2^1} \cdot a_1$	{cost: 1S+1M}
13:	$a_{570} \leftarrow (a_{285})^{2^{285}} \cdot a_{285}$	{cost: 285S+1M}
14:	**return** $(a_{570})^{2^1}$	{cost: 1S}

3.6 Scalar Multiplication

The scalar multiplication over Koblitz curves is to convert a scalar k to a radix τ expansion where $k = \sum u\tau$ and $u \in \{0, +1, -1\}$. After conversion, NAF method is applied to τ-*adic* representation. The τ-*adic* analogue of the ordinary NAF is known as τ-*adic*(TNAF). With extra memory consumption for pre-computation, we can apply a window method named wTNAF. In this paper, we used the 4TNAF method which has window size 4 and the number of addition is reduced to a quarter of the conventional double and add method.

4 Evaluation

For the test over ARMv8 architecture, we set the development environment as follows. We used Xcode (ver 6.3.2) as a development IDE and set the optimization level to `-Ofast`. The target device is iPad Mini2 (iOS 8.4). The iPad Mini2 supports Apple A7 with 64-bit architecture operated in 1.3GHz. In Table 2, the

comparison results of binary field arithmetic, scalar multiplication and ECDH over K-571 curve are drawn. For the binary field multiplication, conventional LD method exploits the series of bit-wise exclusive-or and look-up table access operations not that of NEON instruction sets. On the other hand, the KNV method conducts the eight vectorized 8-bit polynomial multiplication namely `VMULL.P8`. This method conducts multiple data at once so performance is better than LD method. However, the method requires high overheads to combine the eight vector results into one so it acts as a performance bottle neck. In proposed method, we exploit new 64-bit polynomial multiplication namely `PMULL`. This method significantly improves the performance by a factor of 8.3 times than previous works. This is possible because it directly outputs the 128-bit outputs rather than vector form. In this paper, we couldn't compare our results with the methods on same ARMv8 architecture because this is the first ECC implementation over ARMv8. The recent work by [5] only explores the short 128-bit binary field multiplication. For the squaring, traditional table based squaring can accelerate the performance by exploiting the look-up table. With `VMULL.P8` NEON instruction set, we can compute the eight 8-bit polynomial multiplication. The squaring operation does not require to realign the intermediate results so it shows much faster performance than that of multiplication. On ARMv8, we can exploit the `PMULL` and performance is enhanced further by a factor of 2.4. Thanks to high performance binary field multiplication and squaring, Itoh-Tujii method which consists of only multiplication and squaring also shows high performance. Our implementation only needs 31,232 clock cycles and this is 56% better than previous implementations. For scalar multiplication, we measure the performance of unknown point and fixed point. We used 4TNAF method for both implementations. Particularly, fixed point scalar multiplication can avoid the point pre-computation so it shows 8 % better than unknown point. Lastly ECDH agreements including scalar multiplication on both unknown and fixed point are completed within 783,705 clock cycles. This improves the performance by a factor of 4.6.

5 Conclusion

In this paper, we show efficient implementation techniques for K-571 curve on ARMv8. We exploit the novel `PMULL` operation and Karatsuba algorithm to reduce the computation time. Our proposed method improves the performance by a factor of 4.6 times than previous implementations on ARMv7. This is first work targeting the binary field ECC over ARMv8.

References

1. E. Al-Daoud. An improved implementation of elliptic curve digital signature by using sparse elements. *Int. Arab J. Inf. Technol.*, 1(2):203–208, 2004.
2. D. J. Bernstein. Batch binary edwards. In *Advances in Cryptology-CRYPTO 2009*, pages 317–336. Springer, 2009.

Table 2. Comparison results of binary field arithmetic, scalar multiplication and ECDH over K-571

Algorithm	Architecture	Processor	Clock Cycles
Multiplication			
LD [3]	Cortex-A8	ARMv7	3,071
LD [3]	Cortex-A9	ARMv7	3,140
LD [3]	Cortex-A15	ARMv7	1,424
KNV [3]	Cortex-A8	ARMv7	1,506
KNV [3]	Cortex-A9	ARMv7	1,889
KNV [3]	Cortex-A15	ARMv7	1,103
Proposed Method (KNP)	**Apple-A7**	**ARMv8**	**132**
Squaring			
Table [3]	Cortex-A8	ARMv7	349
Table [3]	Cortex-A9	ARMv7	394
Table [3]	Cortex-A15	ARMv7	282
VMULL [3]	Cortex-A8	ARMv7	126
VMULL [3]	Cortex-A9	ARMv7	146
VMULL [3]	Cortex-A15	ARMv7	99
Proposed Method (PMULL)	**Apple-A7**	**ARMv8**	**41**
Inversion (Itoh-Tsujii)			
Previous Method [3]	Cortex-A8	ARMv7	90,936
Previous Method [3]	Cortex-A9	ARMv7	97,913
Previous Method [3]	Cortex-A15	ARMv7	71,220
Proposed Method	**Apple-A7**	**ARMv8**	**31,232**
Scalar multiplication			
Proposed Method (Unknown Point)	**Apple-A7**	**ARMv8**	**408,720**
Proposed Method (Fixed Point)	**Apple-A7**	**ARMv8**	**374,985**
ECDH Agreement			
Previous Method [3]	Cortex-A8	ARMv7	4,870,000
Previous Method [3]	Cortex-A9	ARMv7	6,018,000
Previous Method [3]	Cortex-A15	ARMv7	3,603,000
Proposed Method	**Apple-A7**	**ARMv8**	**783,705**

3. D. Câmara, C. P. Gouvêa, J. López, and R. Dahab. Fast software polynomial multiplication on arm processors using the neon engine. In *Security Engineering and Intelligence Informatics*, pages 137–154. Springer, 2013.
4. S. for Efficient Cryptography Group. Recommended elliptic curve domain parameters. 2000.
5. C. P. Gouvêa and J. López. Implementing gcm on armv8. In *Topics in Cryptology—CT-RSA 2015*, pages 167–180. Springer, 2015.
6. J. Großschädl, R. M. Avanzi, E. Savaş, and S. Tillich. Energy-efficient software implementation of long integer modular arithmetic. In *Cryptographic Hardware and Embedded Systems–CHES 2005*, pages 75–90. Springer, 2005.
7. T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in gf (2 m) using normal bases. *Information and computation*, 78(3):171–177, 1988.
8. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.

9. J. López and R. Dahab. High-speed software multiplication in f2m. In *Progress in CryptologyINDOCRYPT 2000*, pages 203–212. Springer, 2000.
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*, 2015.
13. M. Shirase, Y. Miyazaki, T. Takagi, and D.-G. HAN. Efficient implementation of pairing-based cryptography on a sensor node. *IEICE transactions on information and systems*, 92(5):909–917, 2009.
14. Steve Ranger. Internet of things and wearables drive growth for ARM. Available for download at `http://www.zdnet.com/article/internet-of-things-and-wearables-drive-growth-for-arm/`, Apr. 2014.

Appendix A. Polynomial Addition in Assembly Code

Algorithm 7 571-bit Polynomial Addition

Require: 571-bit Operands A and B.
Ensure: 571-bit Result $C = A \oplus B$.

```
 1: ld1.16b {v0, v1, v2, v3}, [x1], #64
 2: ld1.8b {v4}, [x1]
 3: ld1.16b {v5, v6, v7, v8}, [x2], #64
 4: ld1.8b {v9}, [x2]
 5: eor.16b v5, v5, v0
 6: eor.16b v6, v6, v1
 7: eor.16b v7, v7, v2
 8: eor.16b v8, v8, v3
 9: eor.16b v9, v9, v4
10: st1.16b {v5,v6,v7,v8}, [x0], #64
11: st1.8b {v9}, [x0]
```

Appendix B. Polynomial Squaring in Assembly Code

Algorithm 8 571-bit Polynomial Squaring

Require: 571-bit Operand A.
Ensure: 1142-bit Result $C = A^2$.

```
 1: ld1.16b {v0, v1, v2, v3}, [x1], #64
 2: ld1.8b {v4}, [x1]
 3: pmull v5.1q, v0.1d, v0.1d
 4: pmull2 v6.1q, v0.2d, v0.2d
 5: pmull v7.1q, v1.1d, v1.1d
 6: pmull2 v8.1q, v1.2d, v1.2d
 7: pmull v9.1q, v2.1d, v2.1d
 8: pmull2 v10.1q, v2.2d, v2.2d
 9: pmull v11.1q, v3.1d, v3.1d
10: pmull2 v12.1q, v3.2d, v3.2d
11: pmull v13.1q, v4.1d, v4.1d
12: st1.16b {v5, v6, v7, v8}, [x0], #64
13: st1.16b {v9, v10, v11, v12}, [x0], #64
14: st1.16b {v13}, [x0]
```