Compact Implementations of Public Key Cryptography

No Author Given

No Institute Given

Abstract. In this paper, we present representative public key cryptography including RSA, ECC and Ring-LWE over low-end device namely AVR and high-end device namely ARM-NEON processors. For RSA over AVR, we present a hybrid method to combine Karatsuba multiplication and Montgomery reduction for high speed software implementations. We divided Montgomery reduction into two sub-parts including the one for the conventional Montgomery reduction and the other one for Karatsuba aided multiplication. This approach reduces the multiplication complexity of n-limb Montgomery reduction from $\theta(n^2 + n)$ to asymptotic complexity $\theta(\frac{7n^2}{8} + n)$. The practical implementation results over embedded processors also show performance enhancements by 11%. For RSA over ARM-NEON, we introduce a novel Double Operand Scanning (DOS) method to speed-up multi-precision squaring with non-redundant representations on SIMD architecture. The DOS technique partly doubles the operands and computes the squaring operation without Read-After-Write (RAW) dependencies between source and destination variables. Furthermore, we presented Karatsuba Cascade Operand Scanning (KCOS) multiplication and Karatsuba Double Operand Scanning (KDOS) squaring by adopting additive and subtractive Karatsuba's methods, respectively. The proposed multiplication and squaring methods are compatible with separated Montgomery algorithms and these are highly efficient for RSA crypto system. Finally, our proposed multiplication/squaring, separated Montgomery multiplication/squaring and RSA encryption outperform the best-known results by 22/41%, 25/33% and 30% on the Cortex-A15 platform. For prime field ECC over AVR, we describe the implementation details of field arithmetic for NUMS256, Ted379 and NUMS384. For prime field ECC over ARM-NEON, we present high speed parallel multiplication and squaring algorithms for the Mersenne prime $2^{521} - 1$. Using the approaches, ECDH on NIST's (and SECG's) curve P-521 requires 8.1/4M cycles on an ARM Cortex-A9/A15, respectively. As a comparison, on the same architecture openSSL's ECDH speed test for curve P-521 requires 23.8/18.7M cycles for ARM Cortex-A9/A15, respectively. We exploit 1-level Karatsuba method in order to provide asymtotically faster integer multiplication and multiplication and accumulation operation for efficient reduction algorithm. For binary field ECC over ARM-NEON, 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. For Ring-LWE over AVR, we implement the public-key encryption scheme based on the ring variant of the Learning with Errors (ring-LWE) problem that way proposed by Lyubashevshy et al. Our contributions include the following optimizations: (1) for the Number Theoretic Transform (NTT) based polynomial multiplication, we propose the Montgomery reduction techniques for speeding up the modular coefficient multiplication, (2) for a discrete Gaussian sampler based on the Knuth-Yao (KY) random walk algorithm, we reduce the running memory requirements of with an optimized LUT representation. For medium-term security level, our high-speed optimized ring-LWE implementation requires only $725K$ and $315K$ clock cycles for encryption and decryption. Similarly for long-term security level, the encryption and decryption take $2,492K$ and $640K$ clock cycles, respectively. These achieved results set new speed records for ring-LWE encryption scheme on 8-bit processors. For NTT over ARM-NEON, we implement the Vectorized Number Theoretic Transform (VNTT) computation in parallel way over SIMD architecture. Our contributions include the following optimizations: (1) we vectorized the Iterative Number Theoretic Transform for parallel operations, (2) we propose the 32-bit wise Shifting-Addition-Multiplication-Subtraction-Subtraction (SAMS2) techniques for speeding up the modular coefficient multiplication, (3) we exploit the incomplete arithmetic for representing the coefficient to reduce the number of reduction operations. For medium-term security level, our optimized NTT implementation requires only $27,910$ clock cycles.

Keywords: Public-key cryptography, Modular arithmetic, SIMD-level parallelism, Vector instructions, RSA, Ring learning with errors (Ring-LWE), software implementation, public-key encryption, ARM-NEON, Number Theoretic Transform (NTT), Polynomial Multiplication, Binary Field Multiplication, ARMv7, ARMv8, Koblitz Curve, Elliptic Curve Cryptography, P-521, Karatsuba,

1 RSA Implementation over AVR

Public Key Cryptography (PKC) is still considered computation-intensive, because the underlying arithmetic operations are performed on long integers ranging from hundreds to thousands of bits. Of many arithmetic operations, multi-precision modular arithmetic is a performance-critical building block of both traditional public-key algorithms and elliptic curve crypto-systems. This is in particular the case for the modular multiplication/squaring, which demands careful optimization to achieve acceptable performance. One of the most important modular reduction techniques is Montgomery's algorithm, which was originally introduced in 1985 [55] and has been widely deployed in real-world applications. Montgomery multiplication avoids the expensive trial division operation and performs simple shift operations instead.

In 1996, Koc et al. analyzed the Montgomery multiplication for multi-precision algorithms [38]. Various multiplication methods and combinations are introduced and all of them require $\theta(2n^2 + n)$ and $\theta(n^2 + n)$ for Montgomery multiplication and reduction respectively[1]. After then many works are done to improve performance of Montgomery multiplication with conventional methods [43, 41, 45]. Particularly, the work done by [28] exploits asymptotically faster integer multiplication namely Karatsuba multiplication for multiplication part of Montgomery multiplication. However, they didn't try to reduce the complexity of Montgomery reduction. Since Montgomery reduction has higher complexity over multiplication, any efforts spent on optimizing Montgomery reduction is well spent.

In this section, we present a novel hybrid approach to combine Karatsuba multiplication and Montgomery reduction algorithms. We separated Montgomery reduction into two sub-parts including the conventional Montgomery reduction and Karatsuba aided multiplication parts. This approach reduces the multiplication complexity of n-limb Montgomery reduction from $\theta(n^2+n)$ to asymptotic complexity $\theta(\frac{7n^2}{8} + n)$ and practical implementation results over embedded processors show performance enhancements by 11%.

1.1 Multi-precision Multiplication

Multi-precision multiplication is basic building block of Montgomery multiplication. In this subsection, we explore the features of various multiplication methods. The basic school book approach is Operand Scanning method. In the inner loop, one operand holds a value and computes the partial product by multiplying all the multiplicands. While in the outer loop, the index of the operand increases by a word-size and then the inner loop is executed. The alternative approach is Product Scanning method. The product scanning method computes all partial products in the same column. Since each partial product in the column is computed and then accumulated, additional storages are not needed for intermediate results [14]. In CHES'04, Hybrid Scanning method was introduced. The hybrid scanning method combines both advantages of operand scanning and product scanning. The multiplication is performed on a block scale using product scanning. Inner block partial products follow the operand scanning approach. Therefore, this method reduces the number of memory accesses by sharing the operands within the block [30]. In CHES'11, a fresh multiplication technique named Operand Caching method was introduced. The inner loop of operand caching method follows the product scanning method, but it divides the calculation into several row subsections [33]. By reordering the sequence of inner and outer row subsections, previously loaded operands in general purpose registers are reused for the next partial products. In WISA'12, Consecutive Operand

[1] Number of limb $(n) = \lceil$ Total bit-width (m) / word-width (w) $\rceil$

Algorithm 1 Montgomery reduction

```
Require: An $m$-bit modulus $M$, Montgomery radix $R = 2^m$, an operand $T$ where
    $T = A \cdot B$ or $T = A \cdot A$ in the range $[0, 2^{2m})$, and pre-computed constant $M' =$
    $-M^{-1} \bmod R$
Ensure: Montgomery product ($Z = \mathrm{MonRed}(T, R, M) = T \cdot R^{-1} \bmod M$)
 1: $Q \leftarrow T \cdot M' \bmod R$
 2: $Z \leftarrow (T + Q \cdot M)/R$
 3: if $Z \geq M$ then $Z \leftarrow Z - M$ end if
 4: return Z
```

Caching method was introduced. The consecutive operand caching provides fully cached operands techniques from the beginning to the end [66].

Above approaches provide the instruction set level optimizations. In order to provide asymptotically faster integer multiplication, Karatsuba's multiplication was proposed [36]. Karatsuba's method reduces a multiplication of two n-limb operands to three multiplications, which have a length of $\frac{n}{2}$-limb. These three half-size multiplications can be performed with any multiplication techniques that we covered before (e.g. operand-scanning method, product-scanning method, hybrid-scanning method, operand-caching method [54, 14, 30, 33, 66]). The Karatsuba method is scheduled in a recursive way as well. There are two typical ways to describe Karatsuba's multiplication such as additive Karatsuba and subtractive Karatsuba. Taking the multiplication of n-limb operand A and B as an example, we represent the operands as $A = A_H \cdot 2^{\frac{n}{2}} + A_L$ and $B = B_H \cdot 2^{\frac{n}{2}} + B_L$. The multiplication ($P = A \cdot B$) can be computed according to the following equation when using additive Karatsuba's method:

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

and subtractive Karatsuba's method:

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

Karatsuba's method turns one multiplication of size n into three multiplications and eight additions of size $\frac{n}{2}$. In [6], a variant of Karatsuba's method named refined Karatsuba's method was introduced, which saves one addition operation with a length of $\frac{n}{2}$. Recently, Hutter and Schwabe achieved the speed records on AVR processors (unrolled fashion, 48, 64, 80, 96, 128, 160, 192 and 256-bit) by carefully optimizing the subtractive Karatsuba's multiplication without conditional statements [32].

1.2 Montgomery Multiplication

The Montgomery algorithms were firstly proposed in 1985 [55]. Montgomery algorithms avoid division in modular multiplication by introducing simple shift operations. Given two integers A and B and the modulus M, to compute the product $P = A \cdot B \ mod \ M$ in Montgomery method, original operands A and B

are converted into Montgomery domain, $A' = A \cdot R \ mod \ M$ and $B' = B \cdot R \ mod \ M$. For efficient computations, Montgomery residue R is selected as a power of 2 and constant $M' = -M^{-1} \ mod \ 2^n$ is pre-computed. To compute the product, following three steps are conducted. $T = A \cdot B$, $Q = T \cdot M' \ mod \ 2^n$, $Z = \frac{(T+Q \cdot M)}{2^n}$. The detailed descriptions of Montgomery reduction is available in Algorithm 1. There are many variants of Montgomery methods. Roughly, we can divide the Montgomery multiplication into two categories including separated and integrated versions. Of integrated variants, partly integrating the multiplication and reduction is named as Coarsely Integrated and fully integrating the multiplication and reduction is called as Finely Integrated modes. The detailed methods are as follows.

The Separated Operand Scanning (SOS) method is computing multiplication and reduction operations separately. The multiplication structure is simple school book method but performance is the worst among the variants because operand scanning method frequently accesses memory to load or store intermediate results and operands. Separated Product Scanning (SPS) method is using product scanning method for multiplication and reduction processes separately. Compared to SOS, the required number of registers is small because intermediate results are stored within only three times of word size. The SPS method is a better choice when it comes to register constrained devices. The Karatsuba multiplication is compatible with separated Montgomery multiplication. In [28], a single level of Karatsuba on top of Comba's method (also known as product scanning) is introduced. The Karatsuba approach for 1024-bit modular multiplication can reduce the number of multiplication and addition instructions by a factor 1.14 and 1.18 respectively, than that of sequential interleaved Montgomery approach. In terms of integrated version, Coarsely Integrated Operand Scanning (CIOS) method improves previous SOS method by integrating the multiplication and reduction steps. Instead of computing whole multiplication processes separately, multiplication and reduction steps are alternated in every loop. With this technique we can update intermediate results without additional memory accesses. In case of CIOS, two inner loops are computed but Finely Integrated Operand Scanning (FIOS) integrates the both inner loops of multiplication and reduction and compute the one inner loop. This method reduces intermediate result load and store by computing all results in the intermediate results. Finely Integrated Product Scanning (FIPS) is conducting product scanning multiplication and reduction in integrated model. Since this method does not re-load intermediate results, the method is more efficient than FIOS method. The Coarsely Integrated Hybrid Scanning (CIHS) adopts hybrid multiplication. Firstly half of multiplication is conducted with product scanning, after then multiplication and reduction is coarsely integrated in operand scanning methods. Recently, the work in [41] discussed the performance of different Montgomery multiplications and analyzed the exact computation complexity at the instruction level. They discuss different hybrid Montgomery multiplication algorithms, including Hybrid Finely Integrated Product Scanning (HFIPS), and introduce a novel Hybrid Separated Product Scanning (HSPS). These method finely reschedules inner structure to

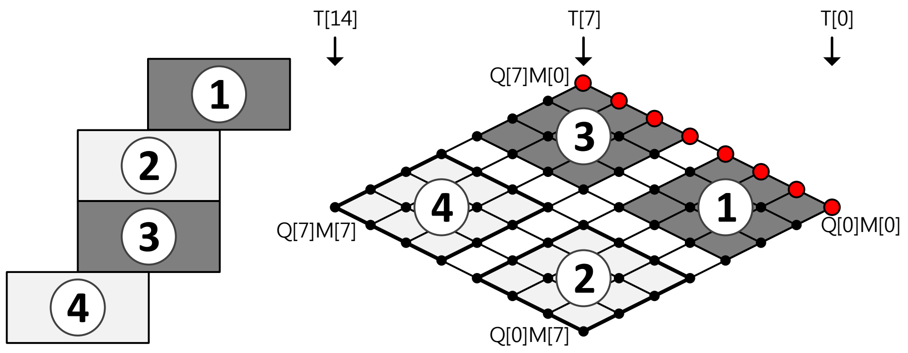

Fig. 1: One-Level Hybrid Montgomery Reduction

reduce number of data transfer instructions. However, there are few software implementations done to adopt Karatsuba multiplication for Montgomery reduction. Since Montgomery reduction has higher complexity than multiplication parts, exploiting Karatsuba algorithm could introduce higher performance than conventional multiplication. In this paper, we introduce a hybrid approach to mix the Karatsuba algorithm and conventional Montgomery reduction for high speed software implementations.

1.3 Proposed Methods

The proposed method divides Montgomery reduction into two sub-Montgomery reduction and two sub-multiplication parts and then each part adopts the most optimized methods. In order to present the asymptotically faster Montgomery reduction, we exploit the Karatsuba multiplication for sub-multiplication parts. Firstly, we conduct the product $Q \leftarrow T \cdot M' \bmod R$ (Step 1 of Algorithm 1) in ordinary way. Secondly the product $Q \cdot M$ is computed in a hybrid way (Step 2). We handle the half of them with product scanning based Montgomery reduction and the other half with Karatsuba multiplication. If the number of limb is even number, we can easily apply the 1 or 2 levels of Karatsuba multiplication. If the number of limb is odd number, we can find the proper approaches in [56]. Since our method follows separated fashion, the hybrid Montgomery reduction is compatible with any other separated versions of Montgomery multiplication. Particularly, the interleaved Montgomery multiplication algorithm is difficult to use asymptotically faster integer multiplication like Karatsuba algorithm [12]. The separated version has additional benefits over interleaved version. Since the method conducts multiplication/squaring and reduction separately, we can exploit the squaring dedicated functions for Montgomery squaring [70, 39]. The overheads of squaring occupies roughly $70 \sim 80\%$ of that of multiplication and for RSA approximately 5/6 of all operations are spent on squaring.

In the following, we describe the proposed method using a multiplication structure and rhombus form described in the Figure 1. Let M and Q be operand

Algorithm 2 Sub-Montgomery reduction

```
Require: An $\frac{m}{2}$-bit half of modulus $M_{HALF}$ where modulus $M$ is $m$-bit, a half of
    Montgomery radix $R_{HALF}$ where Montgomery radix $R$ is $R = 2^m$, an operand
    $T_{HALF}$ in the range $[0, 2^m)$, and pre-computed constant $M'_{HALF} = -M^{-1}$ mod
    $R_{HALF}$
Ensure: Sub-Montgomery product ($\{CARRY, Z\}$ = SubMonRed($T_{HALF}, M_{HALF}$,
    $R_{HALF}$) $= T_{HALF} \cdot R^{-1}_{HALF}$ mod $M$ and quotient $Q$
 1: $Q \leftarrow T_{HALF} \cdot M'_{HALF}$ mod $R_{HALF}$
 2: $\{CARRY, Z\} \leftarrow (T_{HALF} + Q \cdot M_{HALF})/R_{HALF}$
 3: return $\{CARRY, Z\}$, $Q$
```

and quotient with a length of m-bit that are represented by multiple-word arrays. Each operand/quotient is written as follows: $M = (M[n-1], ..., M[2], M[1], M[0])$, $Q = (Q[n-1], ..., Q[2], Q[1], Q[0])$, whereby $n = \lceil m/w \rceil$, and w is the word size. The intermediate result of multiplication $T = A \cdot B$ is twice length of operand A, and represented by $T = (T[2n-1], ..., T[2], T[1], T[0])$. The multiplication structure describes order of partial products from top to bottom and each point in rhombus form represents a multiplication $Q[i] \times M[j]$. The rightmost corner of the rhombus represents the lowest indices (i, $j = 0$), whereas the leftmost represents corner the highest indices (i, $j = n-1$). The lowermost side represents result indices $T[k]$, which ranges from the rightmost corner ($k = 0$) to the leftmost corner ($k = 2n-1$). The black dots represent a partial product of $Q \times M$ and the red dots represent both $T \cdot M'$ mod R and $Q \times M$ procedures.

The numbers over the Figure 1 indicate the order of computations. The rhombus form and multiplication structure are split into four sub-parts. The parts ① and ③ describe the sub-Montgomery reduction and the parts ② and ④ describe the sub-multiplication parts. In part ①, a partial product of $T[0] \times M'$ mod R is conducted and generates a quotient $Q[0]$. After then a partial product of $Q[0] \times M[0]$ is conducted and then added to the intermediate result $T[0]$. This process is iterated from the least significant word to the most significant word of the part. Unlike ordinary Montgomery reduction, we didn't conduct final subtraction to get completely reduced results in sub-Montgomery reduction. Instead, we delayed final subtraction to the very end of the computations and output the parts of intermediate results and quotient ($Q[0 \sim 3]$). The detailed sub-Montgomery reduction is available in Algorithm 2. The Step 1 and 2 follow the original Montgomery reduction in Algorithm 1 but we used the half of original operands to compute the part of Montgomery reduction. Furthermore, we don't conduct final subtraction but we output the incomplete form of intermediate results $\{CARRY, Z\}$ and quotient Q. Unlike the part ①, the part ② does not have a quotient computation ($Q \leftarrow T \cdot M'$ mod R). Therefore, the part ② has a normal form of multiplication with quotient $Q[0 \sim 3]$ and operand $M[4 \sim 7]$. Recent many works successfully accelerate the multiplication with Karatsuba Algorithm ranging from low-end to high-end embedded processors [32, 7, 73], because Karatsuba multiplication can reduce the complexity of multiplication from $\theta(n^2)$ to $\theta(n^{log_2 3})$ for n-limb. For asymptotically faster

Algorithm 3 Hybrid Montgomery reduction

```
Require: An $m$-bit modulus $M$, Montgomery radix $R = 2^m$, and its half radix
    $R_{HALF} = 2^{\frac{m}{2}}$, an operand $T$ where $T = A \cdot B$ or $T = A \cdot A$ in the range $[0, 2^{2m})$
Ensure: Montgomery product $Z = \text{MonRed}(T, R, M) = T \cdot R^{-1} \bmod M$
 1: $\{CARRY_1, Z_L\}$, $Q_L \leftarrow \text{SubMonRed}(T[0, 2^m), M[0, 2^{\frac{m}{2}}), R_{HALF})$
 2: $K_L \leftarrow Q_L \times M[2^{\frac{m}{2}}, 2^m)$
 3: $\{CARRY_2, K_L\} \leftarrow K_L + Z_L + T[2^m, 2^{m+\frac{m}{2}}) \cdot 2^{\frac{m}{2}}$
 4: $\{CARRY_3, Z_H\}$, $Q_H \leftarrow \text{SubMonRed}(K_L, M[0, 2^{\frac{m}{2}}), R_{HALF})$
 5: $CARRY_3 \leftarrow CARRY_3 + CARRY_2$
 6: $K_H \leftarrow Q_H \times M[2^{\frac{m}{2}}, 2^m)$
 7: $\{CARRY_4, K_H\} \leftarrow K_H + Z_H + CARRY_1 + T[2^{m+\frac{m}{2}}, 2^{2m}) \cdot 2^{\frac{m}{2}}$
 8: $\{CARRY_4, K_H\} \leftarrow CARRY_3 \cdot 2^{\frac{m}{2}} + \{CARRY_4, K_H\}$
 9: if $\{CARRY_4, K_H\} \geq M$ then $K_H \leftarrow \{CARRY_4, K_H\} - M$ end if
10: return $K_H$
```

Montgomery reduction, we exploit Karatsuba algorithm for sub-multiplication parts (part ②). After sub-multiplication computations, the results of multiplication and sub-Montgomery reduction are accumulated to the intermediate results (T). This process is iterated in parts ③ and ④. After computation of part ④, we conduct the final subtraction. This process is required to get a fully reduced result in range of $[0, M)$. In order to ensure constant time computations, we adopted the concept of incomplete modular arithmetic [41]. Instead of byte by comparison, it checks z_m bit. If it is set, modulus remains, and otherwise modulus (M) is set to zero by using bit-masking. After then, the intermediate results (Z) is subtracted by modulus (M). Final result is always in the range of $[0, 2^m)$. This incomplete reduction does not introduce any problems in practice because incomplete representation can still be used as operand in a subsequent Montgomery multiplication [78].

In Algorithm 3, the detailed procedures of Hybrid Montgomery reduction is described. In Step 1, sub-Montgomery reduction is conducted by $\frac{m}{2}$-bit with intermediate results $T[0, 2^m)$ and operands $M[0, 2^{\frac{m}{2}})$. This operation outputs the part of reduction results $\{CARRY_1, Z_L\}$ and quotient Q_L. In Step 2, a partial products of Q_L and $M[2^{\frac{m}{2}}, 2^m)$ is conducted. This operation outputs the results K_L. After then intermediate results including K_L, Z_L and $T[2^m, 2^{m+\frac{m}{2}}) \cdot 2^{\frac{m}{2}}$ are accumulated and outputs the results $\{CARRY_2, K_L\}$. The Step 4, 5 and 6 execute same routines of sub-Montgomery reduction and sub-multiplication operations. In Step 7, we conduct additions of intermediate results including K_H, Z_H, $CARRY_1$, and $T[2^{m+\frac{m}{2}}, 2^{2m}) \cdot 2^{\frac{m}{2}}$. This step should conduct the addition of three operands (K_H, Z_H, $CARRY_1$) in the same column bit. However, ordinary instruction set does not support an addition of three operands. In order to resolve this issue, we tried to exploit carry flag from status registers. Firstly we set the temporal variable to $2^w - 1$ where w is the word-width and add the variable ($CARRY_1$). The $CARRY_1$ variable is only either one or zero. If the bit of $CARRY_1$ is set, the carry flag is set after addition. If not, carry flag is not set. This technique can avoid the one time of intermediate result accumulation by $\frac{n}{2}$

Table 1: Comparison of base instructions for Montgomery reduction and multiplication (excluding final subtraction), †: Product Scanning Multiplication + Hybrid Karatsuba Reduction, ‡: Karatsuba Multiplication + Hybrid Karatsuba Reduction.

Algorithm	`mul`	`load`	`store`	`add`
Montgomery reduction:				
OS	$n^2 + n$	$2n^2 + 2n + 1$	$n^2 + 2n + 1$	$4n^2 + 2n$
PS	$n^2 + n$	$2n^2 + 2n$	$2n + 1$	$3n^2 + 6n$
HK	$\frac{7n^2}{8} + n$	$\frac{7n^2}{4} + \frac{21n}{2} + 2$	$\frac{23n}{2} + 4$	$\frac{21n^2}{8} + \frac{25n}{2} + 4$
Montgomery multiplication:				
FIPS	$2n^2 + n$	$4n^2 - n$	$2n + 1$	$6n^2$
SPS	$2n^2 + n$	$4n^2 + 2n$	$4n + 1$	$6n^2 + 6n$
CIOS	$2n^2 + n$	$4n^2 + 5n$	$2n^2 + 3n$	$8n^2 + 4n$
SOS	$2n^2 + n$	$4n^2 + 3n + 1$	$2n^2 + 3n + 1$	$8n^2 + 2n$
CIHS	$2n^2 + n$	$\frac{11n^2}{2} + \frac{7n}{2}$	$3n^2 + 2n$	$9n^2 + 5n$
FIOS	$2n^2 + n$	$3n^2 + 4n$	$n^2 + n$	$8n^2$
KCM	$\frac{7n^2}{4} + n$	$\frac{7n^2}{2} + 11n + 3$	$10n + 1$	$\frac{21n^2}{4} + 8n + 4$
SHKM †	$\frac{15n^2}{8} + n$	$\frac{15n^2}{4} + \frac{21n}{2} + 2$	$\frac{27n}{2} + 4$	$\frac{45n^2}{8} + \frac{25n}{2} + 4$
SHKM ‡	$\frac{13n^2}{8} + n$	$\frac{13n^2}{4} + \frac{33n}{2} + 7$	$\frac{37n}{2} + 3$	$\frac{39n^2}{8} + \frac{33n}{2} + 6$

($\frac{n}{2}$ `add`, $\frac{n}{2}$ `load` and $\frac{n}{2}$ `store`). After intermediate result accumulation in Step 7 and 8, we conduct the final subtraction to get reduced results. With masking the modulus (M) with $CARRY_4$, our implementation conduct constant time final subtraction.

1.4 Result

Analysis of the Algorithms In this subsection, we explore the complexity of proposed Montgomery reduction in terms of addition/subtraction, multiplication and memory accesses such as load and store. The proposed hybrid Montgomery reduction is a combination of two different multiplication techniques for sub-multiplication and sub-reduction parts. For the sub-multiplication, Karatsuba product scanning multiplication is adopted and reduces the number of multiplication. The complexity is $\frac{3n^2}{4}$ `mul`, $\frac{3n^2}{2} + 6n + 1$ `load`, $7n + 1$ `store` and $\frac{9n^2}{4} + 4n + 2$ `add` for n limbs (see [28] for details). For the sub-Montgomery reduction, the product scanning reduction is adopted. Since the method conducts the multiplication in column-wise fashion, Montgomery reduction is efficiently conducted with small number of memory store instructions. The complexity is $n^2 + n$ `mul`, $2n^2 + 2s$ `load`, $2n + 1$ `store` and $3n^2 + 6n$ `add` for n limbs (see [41] for details). The proposed Hybrid Karatsuba Montgomery reduction consists of two $\frac{n}{2}$-limb Karatsuba Product Scanning multiplications and two $\frac{n}{2}$-limb Product Scanning Montgomery Reduction. The two sub-multiplication costs $\frac{3n^2}{8}$ `mul`, $\frac{3n^2}{4} + 6n + 2$ `load`, $7n + 2$ `store` and $\frac{9n^2}{8} + 4n + 4$ `add`. The remaining two sub-Montgomery reduction costs $\frac{n^2}{2} + n$ `mul`, $n^2 + 2n$ `load`, $2n + 2$ `store` and $\frac{3n^2}{2} + 6n$

`add`. In order to accumulate the intermediate results, we need additional instructions including $\frac{5n}{2}$ `load`, $\frac{5n}{2}$ `store` and $\frac{5n}{2}$ `add`. Finally, we can draw the total complexity of Hybrid Karatsuba Montgomery reduction as follows. The costs are $\frac{7n^2}{8}+n$ `mul`, $\frac{7n^2}{4}+\frac{21n}{2}+2$ `load`, $\frac{23n}{2}+4$ `store` and $\frac{21n^2}{8}+\frac{25n}{2}+4$ `add`. For the Montgomery multiplication, we calculated two cases. One is combining conventional product scanning with proposed hybrid Montgomery reduction. The costs are $\frac{15n^2}{8}+n$ `mul`, $\frac{15n^2}{4}+\frac{21n}{2}+2$ `load`, $\frac{27n}{2}+4$ `store` and $\frac{45n^2}{8}+\frac{25n}{2}+4$ `add`. Another combination is using Karatsuba product scanning with proposed hybrid Montgomery reduction. The costs are $\frac{13n^2}{8}+n$ `mul`, $\frac{13n^2}{4}+\frac{33n}{2}+7$ `load`, $\frac{37n}{2}+3$ `store` and $\frac{39n^2}{8}+\frac{33n}{2}+6$ `add`. Finally, we achieved the lowest multiplication complexity such as $\frac{7n^2}{8}+n$ and $\frac{13n^2}{8}+n$ for Montgomery reduction and multiplication, respectively.

Performance Evaluation For practical real world evaluation, we implemented the hybrid Karatsuba Montgomery reduction over 8-bit AVR microcontrollers[2]. The chipset is widely used for low-cost smartcard and wireless sensor nodes. An AVR processor features 32 general-purpose registers, of which six are used for pointers [4]. In particular, the register pair (`R26,R27`) is aliased as `X` pointer, the register pair (`R28,R29`) is aliased as `Y` pointer, and the the register pair (`R30,R31`) is aliased as Z pointer. The AVR microcontrollers have separate memories and buses for program and data a simple single-issue pipeline. It has a total of 133 instructions and each instruction has a fixed latency. Ordinary arithmetic/logical instructions (e.g. `add`) are executed in a single clock cycle, while a `mul` instruction takes two clock cycles, and also `load/store` instructions take two cycles. Most of the software implementation on AVR processors is written in both mixed C and Assembly code. The C function-call ABI specifies that the first three 16-bit arguments (e.g., pointers) are passed in register pairs (`R24, R25`), (`R22,R23`), and (`R21,R20`). It furthermore specifies that registers `R2-R17`, `R28`, and `R29` are "called-saved" registers, and the register `R1` is assumed by the compiler to always contain zero thus has to be set to zero before returning from a C function.

To the best of my knowledge, the most fast Montgomery multiplication on AVR processor is based on Reverse Product Scanning (RPS) [45]. The RPS method processes four bytes of the two operands in each iteration of the inner loop and employs two carry-catcher registers to minimize the number of `add` instructions. The 1024-bit multiplication and Montgomery algorithm only take 102994 and 210139 clock cycles. The interesting point is the program is fully parameterized feature. The program codes can cover whole multiplication and Montgomery multiplication ranging from 160, 192, 224, 256, 384, 512, 768 to 1024-bit. For the short integers, Hutter and Schwabe achieved the speed records on AVR processors (unrolled fashion, 48, 64, 80, 96, 128, 160, 192 and 256-bit)

[2] The proposed method is not limited to the specific target processors because our method presents the lower complexity of multiplication rather than specific target oriented instruction set level optimizations. For this reason, other processors such as MSP, ARM, INTEL can exploit the proposed method.

Algorithm 4 Additive Karatsuba Multiplication

```
Require: An even $m$-bit operands $A(A_{LOW} + A_{HIGH} \cdot 2^{\frac{m}{2}})$, $B(B_{LOW} + B_{HIGH} \cdot 2^{\frac{m}{2}})$
Ensure: $2m$-bit result $C = A \cdot B$
 1: $L = A_{LOW} \cdot B_{LOW}$
 2: $H = A_{HIGH} \cdot B_{HIGH}$
 3: $\{A_{CARRY}, A_{SUM}\} = A_{LOW} + A_{HIGH}$
 4: $\{B_{CARRY}, B_{SUM}\} = B_{LOW} + B_{HIGH}$
 5: $M = A_{SUM} \cdot B_{SUM}$
 6: $M = M + (AND(COM(A_{CARRY}), B_{SUM})) \cdot 2^{\frac{m}{2}}$
 7: $M = M + (AND(COM(B_{CARRY}), A_{SUM})) \cdot 2^{\frac{m}{2}}$
 8: $M = M + (AND(A_{CARRY}, B_{CARRY})) \cdot 2^{\frac{m}{2}}$
 9: $C = L + (M - L - H) \cdot 2^{\frac{m}{2}} + H \cdot 2^m$
10: return $C$
```

by carefully optimizing the subtractive Karatsuba's multiplication without conditional statements [32]. For 256-bit inputs, their software achieved 4797 cycles and this is 20 % faster than previous works. Their short integer multiplication successfully improves the current state of the art Curve25519 implementations [19]. The software computes a Diffie-Hellman shared secret within 13,900,397 clock cycles. While the multiplication implementation is written in fully unrolled way to achieve the best possible performance, it can increase the code size significantly, particularly when the number of digits is large. The possible solution is to partially unrolled the loops. In our implementation, we exploit unrolled 256-bit multiplication as a core operation and call the function recursively for larger than 256-bit digit. The Hutter's 256-bit subtractive Karatsuba multiplication uses 3-levels of Karatsuba. Due to the high register usages of 1 and 2-level Karatsuba blocks, intermediate results and absolute differences are stored in the memory and stack storages. In total, the 256-bit multiplication needs 352 `LD/LDD` instructions, 204 `ST/STD` instructions, 82 `PUSH` instructions, 114 `POP` instructions, 8 `IN` instructions, and 32 OUT instructions. For 512 and 1024-bit Karatsuba multiplication, we used additive Karatsuba's multiplication. The addition, subtraction, two's complement and logical-and operations are written in looped way for small code size and scalability. The additive Karatsuba's multiplication needs to perform several additions and subtractions. The addition of two $\frac{m}{2}$ bit operands (i.e. $A_H + A_L$ and $B_H + B_L$) may generate $(\frac{m}{2} + 1)$-th carry bit. A straightforward carry handling would cause physical vulnerability such as timing attacks [51]. The smart counter measure is "carry-propagated" addition proposed by [45]. The method conducts masking the intermediate results with carry bit. The detailed constant-time additive Karatsuba's multiplication is described in Algorithm 6. Firstly the partial products on $A_{LOW} \cdot B_{LOW}$ and $A_{HIGH} \cdot B_{HIGH}$ are conducted. After then, "carry-propagated" addition is conducted on $A_{LOW} + A_{HIGH}$ and $B_{LOW} + B_{HIGH}$. And then, the middle partial products on $A_{SUM} \cdot B_{SUM}$ are conducted. After then, carry bits including A_{CARRY} and B_{CARRY} are two's complemented (COM) and logical-and operation (AND) is conducted on partial products B_{SUM} and A_{SUM} with the

Table 2: Execution time (in clock cycles) of Montgomery reduction and multiplication implementations for 512 and 1024 bits operands on an ATmega128, †: Product Scanning Multiplication + Hybrid Karatsuba Reduction, ‡: Karatsuba Multiplication + Hybrid Karatsuba Reduction.

Implementation	512	1024
AVR implementations of Montgomery reduction:		
Product-Scanning [41]	29158	111478
Product-Scanning [45]	28265	107144
One-Level Hybrid Karatsuba	**27407**	**97153**
Two-Level Hybrid Karatsuba	**N/A**	**95055**
AVR implementations of Montgomery multiplication:		
Hybrid Finely Integrated Operand Scanning [41]	79760	316018
Hybrid Coarsely Integrated Hybrid Scanning [41]	74435	290549
Hybrid Separated Operand Scanning [41]	69301	268788
Hybrid Coarsely Integrated Operand Scanning [41]	65033	253787
Hybrid Separated Product Scanning [41]	57281	221044
Hybrid Finely Integrated Product Scanning [41]	56339	220596
Hybrid Separated Product Scanning [45]	54396	210139
Separated Hybrid Karatsuba †	**53616**	**197956**
Separated One-Level Hybrid Karatsuba ‡	**46146**	**160070**
Separated Two-Level Hybrid Karatsuba ‡	N/A	**157972**

results of $COM(A_{CARRY})$ and $COM(B_{CARRY})$, respectively. The outputs and the result of $AND(A_{CARRY}, B_{CARRY})$ are added to middle block of intermediate results. Finally, whole partial products including L, M and H are summed up by following equation ($C = L + (M - L - H) \cdot 2^{\frac{m}{2}} + H \cdot 2^m$). From Step 6 to 9, all addition and subtraction operations are conducted. For large integer multiplication, we used multiple level of additive Karatsuba multiplications in recursive way. We used 256-bit Hutter's 3-level of subtractive Karatsuba multiplication as a core multiplication operation and conduct 1- and 2-level of additive Karatsuba multiplication for 512- and 1024-bit multiplications. For sub-Montgomery reduction, we exploit the Zhe's RPS multiplication [45]. This algorithm ensures high performance and scalability among presented Montgomery multiplication. Finally, our hybrid Montgomery reduction exploits two sub-Karatsuba multiplication operations and two sub-RPS Montgomery reduction operations.

In Table 2, execution time of Montgomery reduction and multiplication implementations are available. For 1024-bit Montgomery reduction, we used 512-bit Karatsuba multiplication for two sub-multiplication operations and 512-bit sub-Montgomery reduction operations. For further optimizations, we tried 2-level of Montgomery reduction. In this case, we used two 256-bit and two 512-bit sub-multiplication operations and four 256-bit RPS sub-Montgomery reduction operations[3]. The previous works [41, 45] are done by looped fashion and our method is done by partially looped fashion. In order to provide fair comparison

[3] Detailed descriptions are available in Appendix. C

results, we estimate the performance of fully unrolled previous works. Taking a 1024-bit cases, looped version needs two more registers for outer and inner loop counters. We can use this registers for temporal registers in unrolled version but we cannot increase the width of inner loop. If we increase the width by 1, we need 5 more registers (two for operands, two for intermediate results, one for carry catcher). If we use two registers for operand caching, we can reduce two operands load operations in each outer loop. Previous works have 64 outer loops and 1024 inner loops. Each outer loop saves 8 clock cycles (2 load, 1 counter, 1 comparison, 2 branch) and each inner loop saves 4 clock cycles (1 counter, 1 comparison, 2 branch). Total 4608 clock cycles can be saved and scaled previous result is 102536 clock cycles. However, still our partially looped implementation is faster than fully unrolled previous works by 7 %. In direct comparison, our proposed 1 and 2 levels of Hybrid Karatsuba Montgomery reduction improved performance by 9 % and 11 % than previous results. For Montgomery multiplication, we firstly tested the impacts of Hybrid Karatsuba Montgomery reduction. We used RPS multiplication for multiplication parts and then conduct the Hybrid Karatsuba Montgomery reduction. This case improves the performance by 6.1 %, because Montgomery multiplication consists of multiplication and reduction but we only enhance the reduction part in this case. In second case, we used the additive Karatsuba multiplication and then conduct the Hybrid Karatsuba Montgomery reduction for the highest speed performance. By optimizing the both multiplication and reduction, we achieved the performance enhancements by 20 %.

2 RSA Implementation over ARM-NEON

Multi-precision modular multiplication and squaring are performance-critical building blocks of public-key algorithms (e.g. RSA, ElGamal, DSA and ECC). One of the most famous modular reduction techniques is Montgomery's algorithm which avoids division in modular multiplication and squaring [55]. However, the algorithm is still a computation-intensive operation for embedded processors so it demands careful optimizations to achieve acceptable performance. Recently, an increasing number of embedded processors started to employ Single Instruction Multiple Data (SIMD) instructions to perform massive body of multimedia workloads.

In order to exploit the parallel computing power of SIMD instructions, traditional cryptography software needs to be rewritten into a vectorized format. The most well known approach is a reduced-radix representation for a better handling of the carry propagation [34]. The redundant representation reduces the number of active bits per register. Keeping the final result within remaining capacity of a register can avoid carry propagations. In [11], vector instructions on the CELL microprocessor are used to perform multiplication on operands represented with a radix of 2^{16}. In [29], RSA implementations for the Intel-AVX platform uses 256-bit wide vector instructions and the reduced-radix representation for faster accumulation of partial products. At CHES 2012, Bernstein and

Schwabe adopted the reduced radix and presented an efficient modular multiplication on specific ECC curves. Since the target curves only have low Hamming weight in the least significant bits, modular arithmetics are efficiently computed with multiplication and addition operations. At HPEC 2013, a multiplicand reduction method in the reduced-radix representation was introduced for the NIST curves [61]. However, the reduced-radix representation requires to compute more number of partial products than the non-redundant representation, because it needs more words to store previous radix 2^{32} variables into smaller radix. At SAC'13, Bos et al. flipped the sign of the precomputed Montgomery constant and accumulate the result in two separate intermediate values that are computed concurrently in the non-redundant representation [12]. However, the performance of their implementation suffers from Read-After-Write (RAW) dependencies in the instruction flow. Such dependencies cause pipeline stalls since the instruction to be executed has to wait until the operands from the source registers are available to be read. In [52, 53], product-scanning multiplication over SIMD is introduced. The method computes a pair of 32-bit multiplications at once but it accesses to the same destination column to accumulate the intermediate results in each inner loop, causing high RAW dependencies. At CHES 2014, the ECC implementation adopts 2-level Karatsuba multiplication in the redundant representation. However, as author explained in [7, subsection 1.2.], the redundant representation is not proper choice for the standard NIST elliptic curves. The curves allow easy computation of modular operation in radix 2^{32} rather than reduced representations. At ICISC 2014, Seo et al. introduced a novel 2-way Cascade Operand Scanning (COS) multiplication [72]. This method processes the partial products in a non-conventional order to reduce the number of data-dependencies in the carry propagations from the least to most significant words. The same strategy was applied for 2-way NEON-optimized Montgomery multiplication method, called Coarsely Integrated Cascade Operand Scanning (CICOS) method, which essentially consists of two COS computations, whereby one contributes to the multiplication and the second to the Montgomery reduction.

However, there are still two open interesting topics for Montgomery algorithm on ARM-NEON processors [12, 52, 53, 72]. First, the previous work mainly focused on multiplication not squaring. The overheads of squaring occupies roughly $70 \sim 80\%$ of that of multiplication and for RSA approximately $5/6$ of all operations are spent on squaring. Second, previous methods do not consider Karatsuba multiplication. The GMP multi-precision library switches to one Karatsuba level when it comes to 832-bit inputs but recent SIMD implementations even over 1024- and 2048-bit avoided all use of Karatsuba's method [12, 52, 53, 72]. Since Karatsuba multiplication has nice property that it ensures asymptotic complexity $\theta(n^{log_2 3})$, current implementations can be enhanced by using Karatsuba algorithm. In this paper, we work on these two interesting topics by suggesting a non-redundant Double Operand Scanning (DOS) squaring method and constant-time Karatsuba algorithms for multiplication and squaring. Firstly, the DOS technique partly doubles the operands and computes the squaring operation without Read-After-Write (RAW) dependencies between source and destination variables. Secondly, we present constant-time Karatsuba multiplication and squaring over SIMD architectures. We choose different Karatsuba algorithms for multiplication and squaring to ensure better performance in each operation. Furthermore, SISD and SIMD instructions are properly mix used to reduce latencies. Finally, we present separated KCOS and KDOS Montgomery multiplication and squaring for traditional public key cryptography. Our experimental results show that a Cortex-A15 processor is able to execute SKCOS and SKDOS Montgomery multiplication/squaring with 2048-bit operands in only 19680 and 17584 clock cycles, which are almost 25% and 33% faster than the NEON implementation of Seo et al. (26232 cycles according to [72, Table 2]).

Summary of Research Contributions The main contributions of our work are summarized as the following four points.

1. *Novel Double Operand Scanning approach for efficient implementation of multi-precision squaring on ARM-NEON processors.* When implementing on the Cortex-A15 processor, only 6288 clock cycles are required for squaring at the length of 2048-bit. The result is the fastest implementations published for the identical platform and non-redundant representations. The details of novel approaches can be found in subsection 4.1 and performance comparison with related works can be found in Table 3 and 4.
2. *Fast Constant-time Karatsuba multiplication/squaring for ARM-NEON processors.* Inspired by subtractive Karatsuba multiplication [32] and constant-time Karatsuba algorithms on AVR [46], we proposed constant-time Karatsuba multiplication and squaring on ARM-NEON, which integrate the additive/subtractive Karatsuba algorithms and COS/DOS operations. These carefully chosen methods allow an efficient multiplication and squaring for large integers. The details of novel approaches can be found in subsection 4.2.
3. *Separated Montgomery algorithm for ARM-NEON processors.* In terms of modular multiplication and squaring, we presented separated Montgomery multiplication and squaring. These are compatible with asymptotically faster integer multiplication and squaring algorithms like Karatsuba methods to boost performance significantly. The details of novel approaches can be found in subsection 4.3.
4. *Efficiently implemented cryptographic library for RSA.* As RSA-based schemes are the most widely used asymmetric primitives, enhancements of Montgomery algorithms should be concerned. Thanks to highly optimized modular multiplication and squaring operations, our work only needs 367408 and 14250720 clock cycles for 2048-bit RSA encryption and decryption over A15 processor, respectively. Performance comparison with related works can be found in Table 3 and 4.

2.1 Multi-precision Squaring Methods

Multi-precision squaring can be utilized with ordinary multiplication methods. However, squaring dedicated method has two advantages over the multiplication methods for squaring computations. First, only one operand (A) is required for squaring computations because both operands share same variables. For this reason, we can reduce the number of registers to retain the operands and memory accesses to load operands by about half times. Second, the some part of partial products output the identical partial product results. For example, both partial products $A[i] \times A[j]$ and $A[j] \times A[i]$ output the same results. By taking accounts of the feature, the parts are multiplied once and added twice (i.e. $2 \times A[i] \times A[j]$) to intermediate results to get identical results of naive approaches (i.e. $A[i] \times A[j] + A[j] \times A[i]$). This squaring approach can reduce the number of partial products from n^2 to $\frac{n^2+n}{2}$ whereby $n = \lceil m/w \rceil$, and w and m is the word size and operand length. In the following sub-subsections, we explore the cutting-edge squaring methods over both SISD and SIMD architectures.

Squaring on SISD There are several optimal squaring methods developed by introducing the efficient order of partial products. Lazy-Doubling (LD) method by [40] delays the doubling process to the end of each inner partial product and then double it at once. The method reduces the number of arithmetic operations by conducting doubling computations on accumulated intermediate results. This technique significantly reduces the number of doubling process to one doubling computation per each inner structure of partial products. In INDOCRYPT'13, Sliding-Block-Doubling (SBD) method was introduced [70]. SBD method computes doubling using "1-bit left shifting" operation at the end of duplicated partial product computation. Recently, Karatsuba squaring was introduced [69]. It divides the traditional squaring architecture into two sub-squaring and one sub-multiplication parts. It computes the multiplication part with the subtractive-Karatsuba multiplication and then remaining two squaring parts are conducted with the SBD technique.

However, the advanced SISD based squaring is not compatible with SIMD architecture. The SISD instruction set can readily handle carry bits with status registers but carry-handing over SIMD architecture incurs a number of pipeline stalls in the non-redundant representations. Furthermore SISD approach does not concern about grouping the multiple operands for parallel computations but SIMD approach should concern the alignments of operands and intermediate results. Let's take an example of 512-bit LD squaring over ($A_{0\sim511} \times A_{0\sim511}$) using the 256-bit COS method as an inner loop. The structure consists of three 256-bit wise multiplications ($A_{0\sim255} \times A_{0\sim255}$, $A_{0\sim255} \times A_{256\sim511}$, $A_{256\sim511} \times A_{256\sim511}$). For starter, a computation over $A_{0\sim255} \times A_{0\sim255}$ is conducted. After then, the duplicated part ($A_{0\sim255} \times A_{256\sim511}$) is computed subsequently. While computing the second part, intermediate results of first part should not be mixed with second part because the intermediate results of second part should be doubled but first part does not need doubling process. After doubling the second part, a number of carry propagations from 257th to 768th bit occur to sum both first

($0 \sim 511$) and second ($256 \sim 767$) parts, The alternative approaches including SBD and Karatsuba squaring methods also suffer from same problems. Firstly, they compute the middle part ($A_{0\sim 255} \times A_{256\sim 511}$) with doubling and then conduct other remaining parts ($A_{0\sim 255} \times A_{0\sim 255}$, $A_{256\sim 511} \times A_{256\sim 511}$). As like LD method, the methods generate chains of carry propagations from 257th to 768th bit to sum both intermediate results.

Squaring on SIMD in redundant representations In case of ARM-NEON architecture, the squaring is only considered over the redundant representation for small integers (below 500-bit) of specific ECC implementations [8, 7]. Over the redundant representation, the squaring method is easily established with doubling the operands or intermediate results because the redundant representation can store carry bits into spare capacities in the register. However, long integers such as 2048- or 3072-bit for RSA cryptosystem is not favorable with redundant representations because the number of partial products significantly increase with smaller radix. In addition, the redundant representation needs more number of registers for operands and intermediate results because redundant representations only use the part of the registers to leave spare bits. Since general purpose registers are limited and cannot retain whole variables, a number of memory accesses to store and load the part of variables are required. Actually, this was not concerned in previous ECC implementations [8, 7], because the 2048-bit working registers are sufficient enough to retain 255, 414-bit ECC curve's operands and intermediate results. Furthermore, redundant representations should conduct carry propagations to fit the results into smaller radix if the next operation is multiplication or squaring. This was not big problem in case of scalar multiplication because the scalar multiplication consists of not only multiplication/squaring but also addition/subtraction operations so we can avoid direct radix-fitting and take advantages of lazy radix-fitting. However, in case of RSA, the main exponentiation operation conducts consecutive multiplication or squaring operation so results always conduct radix-fitting to maintain smaller radix for following multiplication and squaring operations. Furthermore as author explained in [7, subsection 1.2.], the standard NIST elliptic curves allow easy computation of modular operation in radix 2^{32} so suitable radix for $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$ in the redundant representation is radix 2^{16} which would cause considerably higher overheads than benefits. In this stance, we need a squaring specialized operation in non-redundant representations for RSA and specific ECC implementations, but still there are no feasible results available in the non-redundant representation over ARM-NEON.

2.2 Karatsuba's Multiplication

One of the multiplication techniques with sub-quadratic complexity is called Karatsuba's multiplication [36]. Karatsuba's method reduces a multiplication of two n-word operands to three multiplications, which have a length of $\frac{n}{2}$ words. These three half-size multiplications can be performed with any multiplication

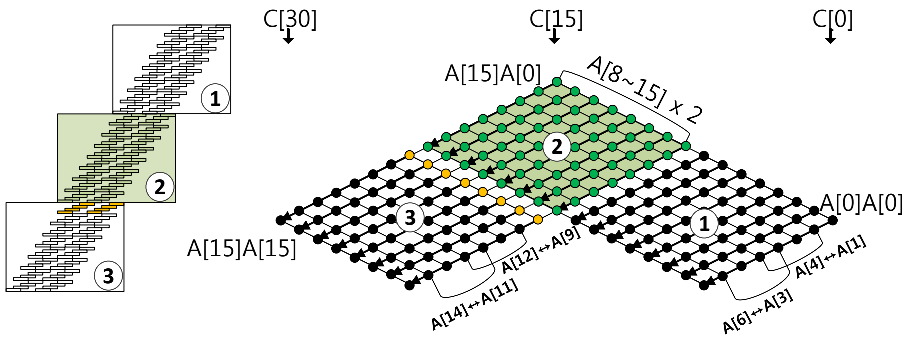

Fig. 2: Double Operand Scanning squaring for SIMD architecture

techniques (e.g. operand-scanning method, product-scanning method, hybrid-scanning method, operand-caching method [54, 14, 30, 33, 66, 67]). The Karatsuba method can also be scheduled in a recursive way and its asymptotic complexity is $\theta(n^{log_2 3})$. There are two typical ways to describe Karatsuba's multiplication such as additive Karatsuba and subtractive Karatsuba. Taking the multiplication of n-word operand A and B as an example, we represent the operands as $A = A_H \cdot 2^{\frac{n}{2}} + A_L$ and $B = B_H \cdot 2^{\frac{n}{2}} + B_L$. The multiplication $(P = A \cdot B)$ can be computed according to the following equation when using additive Karatsuba's method: $A_H \cdot B_H \cdot 2^n + [(A_H + A_L)(B_H + B_L) - A_H \cdot B_H - A_L \cdot B_L] \cdot 2^{\frac{n}{2}} + A_L \cdot B_L$ and subtractive Karatsuba's method: $A_H \cdot B_H \cdot 2^n + [A_H \cdot B_H + A_L \cdot B_L - |A_H - A_L| \cdot |B_H - B_L|] \cdot 2^{\frac{n}{2}} + A_L \cdot B_L$

Karatsuba's method turns one multiplication of size n into three multiplications and eight additions of size $\frac{n}{2}$. In [6], a variant of Karatsuba's method named *refined Karatsuba's method* was introduced, which saves one addition operation with a length of $\frac{n}{2}$. Recently, Hutter and Schwabe achieved the speed records on AVR processors (unrolled fashion, 80, 96, 128, 160, 192 and 256-bit) by carefully optimizing the subtractive Karatsuba's multiplication without conditional statements [32]. The Karatsuba multiplication is readily compatible with non-interleaved Montgomery multiplication. In [28], a single level of Karatsuba on top of Comba's method is introduced. The Karatsuba approach for 1024-bit modular multiplication can reduce the number of multiplication and addition instructions by a factor 1.14 and 1.18 respectively, than that of sequential interleaved Montgomery approach. The Karatsuba approach can enhance Montgomery algorithm with SIMD instructions. However, there are no feasible Montgomery results in non-redundant representations on ARM-NEON engine [12, 52, 53, 72]. In this paper, we present clever approaches to improve Montgomery algorithms with Karatsuba's multiplication in a mixed approach of SISD and SIMD instruction sets.

2.3 Proposed Methods

Double Operand Scanning Squaring Efficient implementation of squaring method is highly relied on computations of duplicated partial products ($A[i] \times A[j]$ and $A[j] \times A[i]$). There are two ways to calculate the duplicated parts of squaring. First approach is doubling the intermediate results [40, 70, 69]. Generally, the method needs to conduct the duplicated part and the other parts separately in order to ensure doubling the duplicated parts except non-duplicated parts. After then, both intermediate results are summed up. However, the addition of both intermediate results causes huge overheads in non-redundant representations by incurring a chain of carry propagations. In order to resolve this issue, we selected a method which doubles operands in advance rather then intermediate results. Proposed Double Operand Scanning (DOS) method uses both doubled and original operands and the duplicated part and non-duplicated parts are computed with different operands (doubled, original) in integrated way and non-redundant representations. This can avoid inefficient carry propagations and a number of pipeline stalls by n times for n word squaring operation compared to the method of doubling the intermediate results.

In Figure 2, we describe DOS squaring for SIMD architecture. The DOS consists of three inner loops (one for duplicated (②) and the other two for non-duplicated parts (①, ③)) and each inner loop follows COS multiplication [72][4]. Taking the 32-bit word with 512-bit squaring as an example, our method works as follows[5]. Firstly, we re-organized operands by conducting transpose operation, which can efficiently shuffle inner vector by 32-bit wise. Instead of a normal order ($(A[0], A[1])$, $(A[2], A[3])$, $(A[4], A[5])$, $(A[6], A[7])$), we classify the operand as groups ($(A[0], A[4])$, $(A[2], A[6])$, $(A[1], A[5])$, $(A[3], A[7])$), for computing two 32-bit wise multiplications where each operand ranges from 0 to $2^{32}-1$ (i.e. `0xffff_ffff` in hexadecimal form). Secondly, multiplication $A[0]$ with

[4] Let A be an operand with a length of m-bit that are represented by multiple-word arrays. Each operand is written as follows: $A = (A[n-1], ..., A[2], A[1], A[0])$, whereby $n = \lceil m/w \rceil$, and w is the word size. The result of multiplication $C = A \cdot A$ is twice length of A, and represented by $C = (C[2n-1], ..., C[2], C[1], C[0])$. For clarity, we describe the method using a multiplication structure and rhombus form. The multiplication structure describes order of partial products from top to bottom and each point in rhombus form represents a multiplication $A[i] \times A[j]$. The rightmost corner of the rhombus represents the lowest indices (i, $j = 0$), whereas the leftmost represents corner the highest indices (i, $j = n-1$). A black arrow over the point indicates the processing of the partial products. The lowermost side represents result indices $C[k]$, which ranges from the rightmost corner ($k = 0$) to the leftmost corner ($k = 2n-1$). Since NEON architecture computes two 32-bit partial products with single instruction, we use two multiplication structures to describe NEON's SIMD operations. These block structures placed in the same level of row represent two partial products with single instruction. In Part 2, the green block and dot represent the partial products with doubled operands; In Part 3, yellow block and dot represent the masked addition with carry bit of doubled operands.

[5] Operands $A[0 \sim 15]$ are stored in 32-bit registers. Intermediate results $C[0 \sim 31]$ are stored in 64-bit registers. We use two packed 32-bit registers in the 64-bit register.

Algorithm 5 Double Operand Scanning Squaring

```
Require: An even $m$-bit operand $A$
Ensure: $2m$-bit result $C = A \cdot A$
1: $C = A_{[0,\frac{m}{2}-1]} \cdot A_{[0,\frac{m}{2}-1]}$
2: $\{A_{CARRY}, A_{DBL[\frac{m}{2},m-1]}\} = A_{[\frac{m}{2},m-1]} \ll 1$
3: $C = C + A_{[0,\frac{m}{2}-1]} \cdot A_{DBL[\frac{m}{2},m-1]} \cdot 2^{\frac{m}{2}}$
4: $C = C + A_{CARRY} \cdot A_{[0,\frac{m}{2}-1]} \cdot 2^{m}$
5: $C = C + A_{[\frac{m}{2},m-1]} \cdot A_{[\frac{m}{2},m-1]} \cdot 2^{m}$
6: return $C$
```

re-organized operands $((A[0], A[4]), (A[2], A[6]), (A[1], A[5]), (A[3], A[7]))$ is computed, generating the partial product pairs including $(C[0], C[4])$, $(C[2], C[6])$, $(C[1], C[5])$, $(C[3], C[7])$ where the results are located from 0 to $2^{64} - 2^{33} + 1$, namely `0xffff_fffe_0000_0001`. Third, partial products are divided into higher bits $(64 \sim 33)$ and lower bits $(32 \sim 1)$ by using transpose operation with 64-bit initialized registers having zero value (i.e. `0x0000_0000_0000_0000`), which outputs a pair of 32-bit results ranging from 0 to $2^{32} - 1$ (i.e. `0xffff_ffff`). After then the higher bits are added to lower bits of upper intermediate results. For example, higher bits of $(C[0], C[4])$, $(C[1], C[5])$, $(C[2], C[6])$, $(C[3])$ are added to lower bits of $(C[1], C[5])$, $(C[2], C[6])$, $(C[3], C[7])$, $(C[4])$. After the addition operation, the least significant word ($C[0]$, lower bits of partial product $(A[0] \times A[0])$) is placed within 32-bit in range of [0, `0xffff_ffff`] and this can be stored into 32-bit wise temporal registers or memory storages. On the other hand, the remaining intermediate results from $C[1]$ to $C[7]$ are placed within [0, `0x1_ffff_fffe`][6], which exceed range of 32-bit in certain cases. However, the addition of intermediate results $(C[1 \sim 7])$ and 32-bit by 32-bit multiplication in next step are placed into 64-bit registers without overflowing, because addition of maximum multiplication result $2^{64} - 2^{33} + 1$ (i.e. `0xffff_fffe_0000_0001`) and intermediate result $2^{33} - 2$ (i.e. `0x1_ffff_fffe`) outputs the final results within 64-bit $2^{64} - 1$ (i.e. `0xffff_ffff_ffff_ffff`)[7]. This process is iterated by 7 times more to complete the first inner loop for partial products $(A[0 \sim 7] \times A[0 \sim 7])$. The intermediate results are retained in temporal registers $((C[8], C[12]), (C[9], C[13]), (C[10], C[14]), (C[11], C[15]))$ placed within $2^{33} - 2$ (i.e. `0x1_ffff_fffe`).

[6] In the first round, the range of result is within [0, `0x1_ffff_fffd`], because higher bits and lower bits of intermediate results $(C[0 \sim 7])$ are located in range of [0, `0xffff_fffe`] and [0, `0xffff_ffff`], respectively. From second round, the addition of higher and lower bits are located within [0, `0x1_ffff_fffe`], because both higher and lower bits are located in range of [0, `0xffff_ffff`].

[7] In the first round, intermediate results $(C[0 \sim 7])$ are in range of [0, `0x1_ffff_fffd`] so results of multiplication and accumulation are in range of [0, `0xffff_ffff_ffff_fffe`]. From second round, the intermediate results are located in [0, `0x1_ffff_fffe`] so results of multiplication and accumulation are in range of [0, `0xffff_ffff_ffff_ffff`].

In second inner loop, we firstly doubled the half of 512-bit operands (256-bit, $A[8 \sim 15]$) by conducting left-shift operation by 1-bit. Since the operation may output 1-bit carry (257th bit), we stored doubled operands into 9 32-bit registers ($A_{CARRY}, A_{DBL}[8 \sim 15]$). Secondly, multiplication $A_{DBL}[8]$ with $(A[0], A[4])$, $(A[2], A[6])$, $(A[1], A[5])$, $(A[3], A[7])$ is computed, generating the partial product pairs including $(C[8], C[12])$, $(C[9], C[13])$, $(C[10], C[14])$, $(C[11], C[15])$. Third, partial products are separated into higher bits ($64 \sim 33$) and lower bits ($32 \sim 1$) by using transpose operation with 64-bit initialized registers having zero value (i.e. `0x0000_0000_0000_0000`). After then the higher bits are added to lower bits of upper intermediate results. After the addition operation, the least significant word is saved into temporal registers or memory storages. This process is iterated by 7 times more to complete the second inner loop for partial products ($A[0 \sim 7] \times A_{DBL}[8 \sim 15]$). The intermediate results are retained in $(C[16], C[20])$, $(C[17], C[21])$, $(C[18], C[22])$, $(C[19], C[23])$ placed within $2^{33} - 2$ (i.e. `0x1_ffff_fffe`).

In third inner loop, we firstly conduct the carry handling by masking the operands with the carry bit (A_{CARRY}) to ensure secure against side channel attacks. By using multiplication and accumulation operation (`VMLAL`), we can multiply the operand with the carry bit (A_{CARRY}) and then the results are added to intermediate results simultaneously. If the carry bit is set, operands $A[0 \sim 7]$ are added to the intermediate results ($(C[16], C[20])$, $(C[17], C[21])$, $(C[18], C[22])$, $(C[19], C[23])$) and otherwise zero values are added to the intermediate results. After then the intermediate results are separated into higher and lower 32-bit wise and added to $(C[17], C[21])$, $(C[18], C[22])$, $(C[19], C[23])$, $(C[20], C[24])$. Secondly, we re-organized operands ($A[8 \sim 15]$) by conducting transpose operation into $(A[8], A[12])$, $(A[10], A[14])$, $(A[9], A[13])$, $(A[11], A[15])$. Thirdly, multiplication $A[8]$ with re-organized operands ($(A[8], A[12])$, $(A[10], A[14])$, $(A[9], A[13])$, $(A[11], A[15])$) is computed, generating the partial product pairs including $(C[16], C[20])$, $(C[17], C[21])$, $(C[18], C[22])$, $(C[19], C[23])$. After then the intermediate results are separated into each higher and lower 32-bit wise and added to $(C[17], C[21])$, $(C[18], C[22])$, $(C[19], C[23])$, $(C[20], C[24])$. After the addition operation, the least significant word is saved into temporal registers or memory storages. This process is iterated by 7 times more to complete the third inner loop for partial products ($A[8 \sim 15] \times A[8 \sim 15]$).

After three inner loops, the results from $C[0]$ to $C[23]$ are perfectly fitted into 32-bit wise word, because the least significant word is outputted in 32-bit way in every round. However, remaining intermediate results ($C[24] \sim C[31]$) are not placed within 32-bit so we should process a chain of carry propagations to satisfy the radix 2^{32}, namely final alignment. The final alignment executes carry propagations on results from $C[24]$ to $C[31]$ to fit into radix 2^{32} with sequential addition and transpose instructions. This process causes pipeline stalls by 8 times, because higher bits of former results are directly added to next intermediate results. In order to reduce these latencies of final alignments, we used SISD rather than SIMD instruction because SISD has lower latencies than SIMD in terms of sequential operations. Finally, 512-bit DOS squaring

Algorithm 6 Additive Karatsuba Multiplication on SIMD

```
Require: An even $m$-bit operands $A(A_{LOW} + A_{HIGH} \cdot 2^{\frac{m}{2}})$, $B(B_{LOW} + B_{HIGH} \cdot 2^{\frac{m}{2}})$
Ensure: $2m$-bit result $C = A \cdot B$
 1: $L = A_{LOW} \cdot B_{LOW}$ (SIMD)
 2: $H = A_{HIGH} \cdot B_{HIGH}$ (SIMD)
 3: $\{A_{CARRY}, A_{SUM}\} = A_{LOW} + A_{HIGH}$
 4: $\{B_{CARRY}, B_{SUM}\} = B_{LOW} + B_{HIGH}$
 5: $M = A_{SUM} \cdot B_{SUM}$ (SIMD)
 6: $M = M + (AND(COM(A_{CARRY}), B_{SUM})) \cdot 2^{\frac{m}{2}}$
 7: $M = M + (AND(COM(B_{CARRY}), A_{SUM})) \cdot 2^{\frac{m}{2}}$
 8: $M = M + (AND(A_{CARRY}, B_{CARRY})) \cdot 2^{\frac{m}{2}}$
 9: $C = L + (M - L - H) \cdot 2^{\frac{m}{2}} + H \cdot 2^{m}$
10: return $C$
```

requires SIMD instructions including 103 `VTRN`, 100 `VMULL/VMLAL`, 100 `VEOR`, 104 `VADD`, 24 `VEXT`, 4 `VSHR/VSHL` and 9 `VMOV` and several SISD `ADDS/ADCS` instructions. The algorithm of DOS squaring is drawn in Algorithm 5. In Step 1, multiplications on $A_{[0,\frac{m}{2}-1]} \times A_{[0,\frac{m}{2}-1]}$ are conducted and stored into results (C). In Step 2, operands $A_{[0,\frac{m}{2}-1]}$ are doubled to output doubled operands ($A_{CARRY}, A_{DBL[\frac{m}{2},m-1]}$). The part of doubled operands ($A_{DBL[\frac{m}{2},m-1]}$) is multiplied by $A_{[0,\frac{m}{2}-1]}$ and added to results in Step 3. In Step 4, the carry bit (A_{CARRY}) is multiplied by $A_{[0,\frac{m}{2}-1]}$ and added to results. In Step 5, multiplications on $A_{[\frac{m}{2},m-1]} \times A_{[\frac{m}{2},m-1]}$ are conducted and then added to intermediate results. Finally, the results are returned in Step 6.

Additive Karatsuba Multiplication The additive Karatsuba's multiplication needs to perform several additions and subtractions (see subsection 3). Among them, the addition of two $\frac{m}{2}$ bit operands (i.e. $A_H + A_L$ and $B_H + B_L$) may generate $(\frac{m}{2} + 1)$-th carry bit. A straightforward carry handling would cause physical vulnerability such as timing attacks [51]. The smart counter measure is "carry-propagated" addition proposed by [46]. The method conducts masking the intermediate results with carry bit. However, under non-redundant representations, addition operation causes a chain of carry propagations. In order to avoid these latencies, we used SISD instructions for the sequential addition and subtraction operations. The detailed constant-time additive Karatsuba's multiplication is described in Algorithm 6. The partial products on $A_{LOW} \cdot B_{LOW}$ and $A_{HIGH} \cdot B_{HIGH}$ are conducted by following COS multiplication for SIMD architecture. After then, "carry-propagated" addition is conducted on $A_{LOW} + A_{HIGH}$ and $B_{LOW} + B_{HIGH}$ with SISD instructions. And then, the middle partial products on $A_{SUM} \cdot B_{SUM}$ are conducted with COS multiplication with SIMD instructions. After then, carry bits including A_{CARRY} and B_{CARRY} are two's complemented (COM) and logical-and operation (AND) is conducted on partial products B_{SUM} and A_{SUM} with the results of $COM(A_{CARRY})$ and $COM(B_{CARRY})$, respectively. The outputs and the result of $AND(A_{CARRY}, B_{CARRY})$ are added to middle block of intermedi-

Algorithm 7 Subtractive Karatsuba Squaring on SIMD

```
Require: An even $m$-bit operand $A(A_{LOW} + A_{HIGH} \cdot 2^{\frac{m}{2}})$
Ensure: $2m$-bit result $C = A \cdot A$
1: $L = A_{LOW} \cdot A_{LOW}$ (SIMD)
2: $H = A_{HIGH} \cdot A_{HIGH}$ (SIMD)
3: $\{A_{BORROW}, A_{DIFF}\} = A_{LOW} - A_{HIGH}$
4: $A_{DIFF} = XOR(A_{BORROW}, A_{DIFF})$
5: $A_{DIFF} = A_{DIFF} + COM(A_{BORROW})$
6: $M = A_{DIFF} \cdot A_{DIFF}$ (SIMD)
7: $C = L + (L + H - M) \cdot 2^{\frac{m}{2}} + H \cdot 2^m$
8: return $C$
```

ate results. Finally, whole partial products including L, M and H are summed up by following equation ($C = L + (M - L - H) \cdot 2^{\frac{m}{2}} + H \cdot 2^m$). From Step 6 to 9, all addition and subtraction operations are conducted sequentially by using SISD operations. The combinations of SISD and SIMD instruction sets reduce the pipeline stalls and latencies. For large integer multiplication, we used multiple level of additive Karatsuba multiplications. For scalability, our Karatsuba multiplications are performed in recursive way. We used 256-bit COS multiplication as a basic multiplication operation and conduct 1-, 2- and 3-level of additive Karatsuba multiplication for 512-, 1024- and 2048-bit multiplications.

Subtractive Karatsuba Squaring For multi-precision squaring, we selected subtractive Karatsuba algorithm. The subtractive Karatsuba algorithm has one advantage over additive Karatsuba algorithm when it comes to squaring. The fact that the partial products on differences of operand ($A_{DIFF} \cdot A_{DIFF}$) always produce non-negative results (M). Thanks to this feature, we can avoid checking the sign of results (M) [46]. As like additive Karatsuba method, we conducted main squaring computations with parallel DOS squaring by using SIMD instructions and the other operations with SISD operations in sequential way. The detailed constant-time subtractive Karatsuba's squaring is described in Algorithm 7. The partial products on $A_{LOW} \cdot A_{LOW}$ and $A_{HIGH} \cdot A_{HIGH}$ are conducted by following DOS squaring. After then, A_{LOW} is subtracted by A_{HIGH} to output the $\{A_{BORROW}, A_{DIFF}\}$. If borrow occurs, the A_{BORROW} is set to $2^{32} - 1$ (i.e. `0xffff_ffff`) and A_{DIFF} is negative value. Otherwise, A_{BORROW} is set to zero (i.e. `0x0000_0000`) and A_{DIFF} is positive value. In order to ensure the differences in positive form, we conduct masking operation with A_{BORROW} variables. Firstly, bit-wise exclusive-or operation (XOR) is conducted on differences (A_{DIFF}) with A_{BORROW}. Secondly, two's complement operation (COM) is conducted on A_{BORROW} and the output is added to the difference (A_{DIFF}). If A_{BORROW} is set to $2^{32}-1$ (i.e. `0xffff_ffff`), A_{DIFF} is two's complemented and otherwise A_{DIFF} maintains its own value. This masking technique is conducted sequentially by using SISD instructions. After then, the middle partial product (M) on $A_{DIFF} \cdot B_{DIFF}$ is conducted with DOS squaring for SIMD instruction sets. Finally, whole partial products including L, M and H are summed up by

Algorithm 8 Calculation of the Montgomery reduction

```
Require: An odd $m$-bit modulus $M$, Montgomery radix $R = 2^m$, an operand $T$ where
    $T = A \cdot B$ or $T = A \cdot A$ in the range $[0, 2M - 1]$, and pre-computed constant
    $M' = -M^{-1} \bmod R$
Ensure: Montgomery product $Z = \text{MonRed}(T, R) = T \cdot R^{-1} \bmod M$
 1: $Q \leftarrow T \cdot M' \bmod R$
 2: $Z \leftarrow (T + Q \cdot M)/R$
 3: if $Z \geq M$ then $Z \leftarrow Z - M$ end if
 4: return Z
```

following equation ($C = L + (L + H - M) \cdot 2^{\frac{m}{2}} + H \cdot 2^m$) with SISD instruction sets. We used 512-bit DOS squaring as a basic squaring operation and conduct multiple Karatsuba squaring in recursive way. For 1024- and 2048-bit squaring implementations, we adopted 1- and 2-level of subtractive Karatsuba squaring operations, respectively.

Separated Karatsuba Cascade/Double Operand Scanning for Montgomery Multiplication and Squaring In [72, 12], the integrated Montgomery multiplication methods are proposed. However, the interleaved version is not compatible with Karatsuba's methods because Karatsuba multiplication recursively conducts the part of partial products but Montgomery reduction is normally performed in sequential way from the least to most significant bits. In order to exploit nice properties of Karatsuba approaches, we selected the separated (non-interleaved) Montgomery algorithm. In the Algorithm 8, we firstly compute multi-precision multiplication ($A \times B$) or squaring ($A \times A$) with KCOS multiplication or KDOS squaring, respectively. After then the intermediate results (T) are multiplied by inverse of modulus (M') and the results are reduced by R and stored into Q. After then, following equation ($(T + Q \times M)/R$) is conducted.

Finally, the calculation of the Montgomery multiplication may require a final subtraction of the modulus (M) to get a fully reduced result in range of $[0, M)$. In order to get the reduced results, the final subtraction is conducted. The operation is computable with conditional branch by checking the carry bit. However, this method has two drawbacks. First two operands should be compared byte by byte via the compare function and the attacker can catch the leakage information because conditional statements consumes different clock cycles [77]. In order to resolve this problem, in [41], author suggested without conditional branch method for Montgomery multiplication. Based on the concept of incomplete modular arithmetic, we don't compare exact value between Z and M, but we use most significant bit (z_m) of Z. If z_m is set, modulus remains, and otherwise modulus (M) is set to zero by using bit-masking. After

Table 3: Results of multiplication/squaring and Montgomery multiplication/squaring and RSA operations in clock cycles on ARM Cortex-A9 platform, *: estimated results

Bit	Cortex-A9					
	Our	[72]	NEON[12]	ARM[12]	GMP[23]	OpenSSL[60]
Multiplication						
512	**1048**	1050	-	-	2176	-
1024	**3791**	4298	-	-	6256	-
2048	**13736**	17080	-	-	19618	-
Squaring						
512	**850**	-	-	-	1343	-
1024	**3315**	-	-	-	4063	-
2048	**9180**	-	-	-	14399	-
Montgomery Multiplication						
512	**2210**	2254	5236	3175	-	-
1024	**8245**	8358	17464	10167	-	-
2048	**30940**	32732	63900	36746	-	-
Montgomery Squaring						
512	**1938**	-	-	-	-	-
1024	**7837**	-	-	-	-	-
2048	**26860**	-	-	-	-	-
RSA encryption						
1024	**156502**	167160*	379736	245167	214064	294831
2048	**535020**	654640*	1358955	872468	791911	1029724
RSA decryption						
1024	**2965820**	-	7166897	4233862	-	4896000
2048	**20977660**	-	47205919	27547434	-	33134700

then, the intermediate results (Z) is subtracted by modulus (M)[8]. Final result may not be the at least non-negative residue but this is always in the range of $[0, 2^m)$. This incomplete reduction does not introduce any problems in practice because incomplete representation can still be used as operand in a subsequent Montgomery multiplication [78].

2.4 Results

Target Platform The ARM Cortex-A9 and A15 series are full implementations of the ARMv7 architecture including NEON engine. Register sizes are 64-bit and 128-bit for double(`D`) and quadruple(`Q`) word registers, respectively. Each register provides short bit size computations such as 8-bit, 16-bit, 32-bit and 64-bit. This feature provides more precise operation and benefits to various

[8] In order to reduce the latencies, we conducted final subtraction with SISD instruction sets because the SISD instruction set provides borrow bits and short delays per instructions rather than that of SIMD.

Table 4: Results of multiplication/squaring and Montgomery multiplication/squaring and RSA operations in clock cycles on ARM Cortex-A15 platform, *: estimated results

Bit	Cortex-A15						
	Our	[72]	[52, 53]	NEON[12]	ARM[12]	GMP[23]	OpenSSL[60]
Multiplication							
512	**640**	658	-	-	-	1184	-
1024	**2464**	2810	-	-	-	4352	-
2048	**8320**	10672	-	-	-	13632	-
Squaring							
512	**516**	-	-	-	-	928	-
1024	**1856**	-	-	-	-	3040	-
2048	**6288**	-	-	-	-	11600	-
Montgomery Multiplication							
512	**1408**	1485	4206	2473	2373	-	-
1024	**5392**	5600	14051	8527	8681	-	-
2048	**19680**	26232	50265	33441	33961	-	-
Montgomery Squaring							
512	**1280**	-	-	-	-	-	-
1024	**4784**	-	-	-	-	-	-
2048	**17584**	-	-	-	-	-	-
RSA encryption							
1024	**95264**	112000*	281020*	207647	195212	152432	224624
2048	**367408**	524640*	1005300*	712542	725336	654240	763120
RSA decryption							
1024	**1957120**	-	-	3332262	3288177	-	3625600
2048	**14250720**	-	-	22812040	23177617	-	24240000

word size computations. The Cortex-A9 processor is adopted in several devices including iPad 2, iPhone 4S, Galaxy S2, Galaxy S3, Galaxy Note 2, PandaBoard and Kindle Fire. The Cortex-A15 is used in Chromebook, NEXUS 10, Tegra 4, Odroid-XU, Galaxy S4 and Galaxy S5.

Evaluation We prototyped our methods for ARM Cortex-A9 and A15 processors, which are equivalent to the target processors used in previous works [72, 12, 52, 53]. We compared our results with best previous results from proceeding version of Seo et al.'s paper presented at ICISC 2014 [72]. In Table 3 and 4, we categorize the timings with respect to the architecture that served as experimental platform[9]. In the case of 2048-bit multiplication, we achieve an execution time of 13736 and 8320 clock cycles on the Cortex-A9 and A15 series, while Seo et al.'s SIMD implementation requires 17080 and 10672 clock cycles. Previous works did not provide a squaring specialized method with non-redundant rep-

[9] We only employ single core and optimization level is set to -O3.

resentations on ARM-NEON [72, 12, 52, 53]. We compared our squaring to the latest GNU multiple precision arithmetic library (GMP) ver 6.0.0a [23]. We compute the 2048-bit squaring in an execution time of 9180 and 6288 clock cycles for A9 and A15, while GMP implementation requires 14399 and 11600 clock cycles. In the case of 2048-bit Montgomery multiplication, we achieve an execution time of 30940 clock cycles on the Cortex-A9 series, while Seo et al.'s SIMD implementation requires 32732 clock cycles. Furthermore, on a Cortex-A15, we compute a 2048-bit Montgomery multiplication within 19680 clock cycles rather than 26232 clock cycles as specified in [72, Table 2]. The Montgomery squaring shows much more optimized results than Montgomery multiplication. The strength of Montgomery squaring is vividly seen in RSA encryption and decryption, because exponentiation operation requires a number of modular squaring[10]. For this reason, our 2048-bit RSA encryption only requires 535020 and 367408 clock cycles, while Seo et al.'s work needs 654640 and 524640 clock cycles. Thus, our work outperforms Seo et al.'s work by approximately 18% and 30% on a Cortex-A9 and Cortex-A15, respectively. For comparison with 2048-bit RSA encryption and decryption of OpenSSL 1.0.2 [60], our implementations are roughly two times faster than that of OpenSSL. Proposed methods satisfy the operand scalability and Karatsuba algorithm under non-redundant representations (see Table 5). In terms of scalability, we can conduct various length of modular multiplication or squaring in a single code by altering loop counter. This would be beneficial for practical usages such as modular operations for random prime numbers. The interesting point is our work even defeats the unrolled work by [72]. In case of Karatsuba algorithm, this is a novel approach to improve SIMD based multiplication and squaring under non-redundant representations. Following are reasons for the significant speed-up compared to Seo et al.'s NEON implementations.

First, we used squaring dedicated method which can compute squaring more efficiently than ordinary multiplication approach. Second, constant-time Karatsuba algorithm is adopted to multiplication and squaring, which provides asymptotically fast methods than traditional approaches. Finally, we properly mix-used SISD and SIMD instruction sets in order to reduce latencies from a number of pipeline stalls.

Comparison to GMP The most well known multiple precision arithmetic library is GMP. The GMP also uses asymptotically fast Karatsuba algorithm. We compared the performances on different long integers ranging from 512-bit to 8192-bit and the comparison graphs are drawn in Figure 3 for ARM Cortex-A9 and A15. For multi-precision multiplication and squaring, our KCOS and KDOS methods show huge performance enhancements in 512-bit by $46 \sim 50\%$ and $37 \sim 44\%$. As length of operand increases, the performance enhancements decrease but we still have high improvements in 8192-bit by $20 \sim 24\%$ and $23 \sim 35\%$, respectively.

[10] RSA benchmark setting: (1) decryption with Chinese Remainder Theorem algorithm, (2) RSA operations with no padding, (3) specialized squaring routine, (4) the public exponentiation $(2^{16} + 1)$, (5) using window method

Table 5: Comparison of proposed implementations with related works

Implementation	Speed Record	Program Style	Scalability	Karatsuba
Published modular multiplication implementations:				
Martins et al. [52, 53]		looped/parameterised	$\surd$	
Bos et al. [12]		looped/parameterised	$\surd$	
Seo et al. [72]		unrolled		
This work (mul)	$\surd$	looped/parameterised	$\surd$	$\surd$
Published modular squaring implementations:				
This work (sqr)	$\surd$	looped/parameterised	$\surd$	$\surd$

3 ECC Implementation over AVR

In this section, we introduce the ECC implementation results on the following notation:

- NUMS256: the prime is $p_{256} = 2^{256} - 189$.
- Ted379: the prime is $p_{379} = 2^{379} - 19$.
- NUMS384: the prime is $p_{384} = 2^{384} - 317$.

3.1 Multiplication and Squaring

We perform the modular multiplication and squaring in a septated fashion, namely, the reduction is executed after the computation of multi-precision multiplication/squaring.

256-bit Multiplication and Squaring We use the 256-bit Karatsuba multi-precision multiplication and squaring.

379/384-bit Multiplication and Squaring As shown in Algorithm 9 and Algorithm 10, we use constant-time subtractive Karatsuba method to split a 384-bit multiplication/squaring into three 192-bit multiplications/suqarings.

For the sake of different trade-off between performance and memory consumption, we combine the Karatsuba method with the state-of-the-art technique for the computation of 384-bit multiplication and squaring as shown in Table 7.

- **Speed-optimized**. We employ Hutter-Schwabe's Karatsuba implementation to perform the 192-bit multiplication, while the squaring is realized by the optimized Karatsuba squaring. That is to say, we actually use three levels Karatsuba method for implementation of 384-bit multiplication and squaring, the first level is implemented in ANSI C, while the second and third levels are implemented in Assembly. The best performance we can achieve for 384-bit multiplication and squaring are 11,898 and 8,037 clock cycles, while 5457 and 3800 bytes are required, respectively. However, the *addition,*

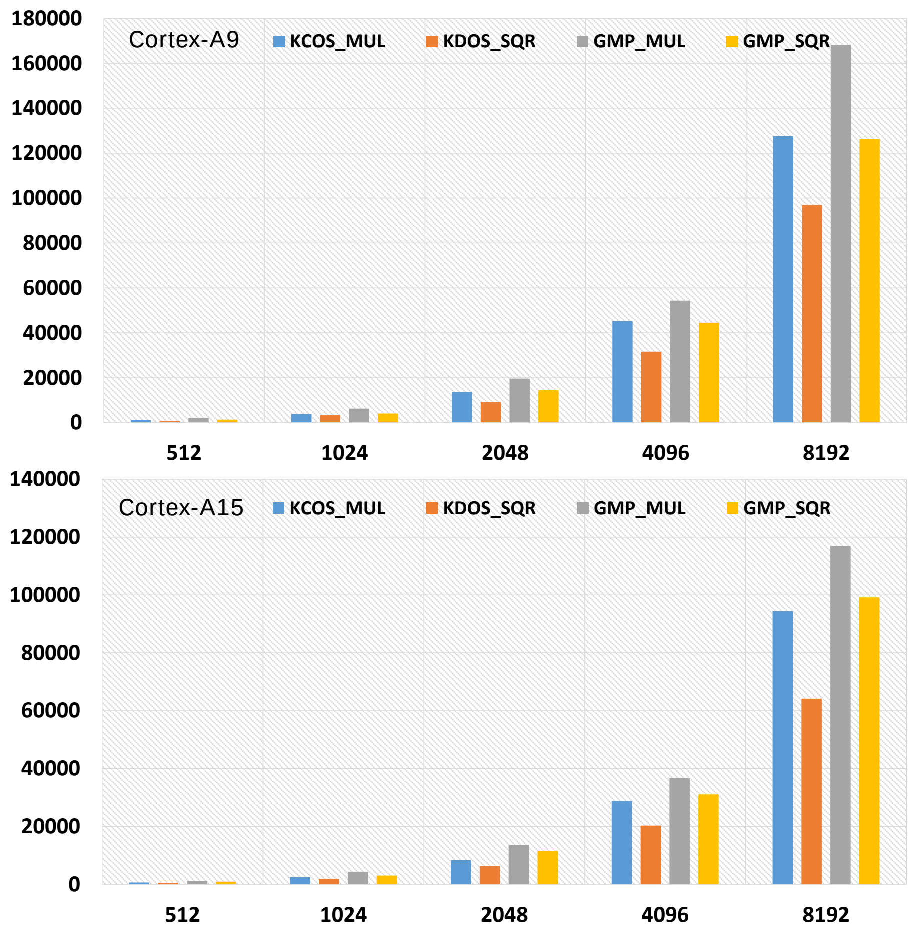

Fig. 3: Results of multiplication/squaring in clock cycles on ARM Cortex-A9/A15

subtraction, negation functions are implemented in looped version, thus, it is possible to further improve the performance of multiplication and squaring by *unrolling the loop*.

– **Memory-optimized**. It is possible to reduce the code size within 2k bytes when using the Karatsuba-Reverse-Product-scanning (KRPS) method. In this case, the KRPS is 18% slower for performance but gain a saving of 2/3 for code size; the RPS squaring is 5% slower for performance but requires only 1/5 for code size. A concrete comparison is given in Table 6.

Fast Incomplete Reduction

Algorithm 9 Constant-time Karatsuba multiplication (subtractive fashion)

```
Require: Two 384-bit operand $A = AH \cdot 2^{192} + AL$ and $B = BH \cdot 2^{192} + BL$.
Ensure: The product $Z = A \cdot B$.
1: Compute abstract value of $ADIFF = |AH - AL|$.    {int_neg; int_sub}
2: Compute abstract value of $BDIFF = |BH - BL|$.    {int_neg; int_sub}
3: Compute abstract value of $ZDIFF = |ADIFF \cdot BDIFF|$.
                                                     {int_neg; karatsuba_mul_192 }
4: Compute $ZL = AL \cdot BL$.                       {karatsuba_mul_192 }
5: Compute $ZH = AH \cdot BH$.                       {karatsuba_mul_192 }
6: Compute $Z = ZH \cdot 2^{384} + (ZL + ZH - ZDIFF) \cdot 2^{192} + ZL$.
                                                     {Three int_add; int_add_word}
7: return $Z$
```

Algorithm 10 Constant-time Karatsuba squaring (subtractive fashion)

```
Require: Two 384-bit operand $A = AH \cdot 2^{192} + AL$.
Ensure: The product $Z = A^2$.
1: Compute abstract value of $ADIFF = |AH - AL|$.    {int_neg; int_sub}
2: Compute abstract value of $ZDIFF = ADIFF^2$.      {karatsuba_sqr_192 }
3: Compute $ZL = AL^2$.                              {karatsuba_sqr_192 }
4: Compute $ZH = AH^2$.                              {karatsuba_sqr_192 }
5: Compute $Z = ZH \cdot 2^{384} + (ZL + ZH - ZDIFF) \cdot 2^{192} + ZL$.
                                                     {Two int_add; int_sub, int_add_word}
6: return $Z$
```

Table 6: Execution time and code size of 384-bit multiplication and squaring

Integer	Combination	Time (cycles)	ROM (bytes)	Karatsuba?
384-bit MUL	(1) + (3)	11898	5474	3 levels
384-bit SQR	(2) + (4)	8,037	3800	3 levels
384-bit MUL	(1) + (5)	14382	1704	1 level
384-bit SQR	(6)	8,505	832	No

(1): Constant-time subtractive Karatsuba multiplication (in ANSI C).

(2): Constant-time subtractive Karatsuba squaring (in ANSI C).

(3): 192-bit Karatsuba multiplication (in Asembly).

(4): 192-bit Karatsuba squaring (in Assembly).

(5): Optimized RPS multiplication (in Assembly).

(6): Optimized RPS squaring (in Assembly).

$2^{379} - 19$. It is easy to observe that

$$2^{379} \equiv 19 \bmod p_1. \tag{3}$$

A straightforward reduction is to repeatedly replace $ZH \cdot 2^{379}$ with $ZH \cdot 19$. However, since 379 is not an 8-bit friendly number (i.e. a multiple of 8), thus, this method requires a lot of shifting operations.

Our fast reduction for $2^{379} - 19$ is inspired by the observation that

$$2^{379} \equiv 19 \Longrightarrow 2^{384} \equiv 608 \bmod p_1. \tag{4}$$

The main idea is to perform the first round reduction with $2^{384} \equiv 608 \bmod p_1$ and then do the second round reduction with $2^{379} \equiv 19 \bmod p_1$. Instead of doing a complete reduction with the final result in the range of $[0, 2^{379} - 19]$, we implement the reduction in an incomplete way which allows the reduction result to stay in the range of $[0, 2^{380}]$. The fast reduction algorithm shown in Algorithm 11 can be described as follows:

1. We perform the first reduction (line 1), i.e. the computation of $(c', Z') = ZH \cdot c1 + ZL$. Instead of computing the whole $ZH \cdot c + ZL$, we only compute $T[0 \sim 5] = ZH[44 \sim 47] \cdot c + ZL[44 \sim 47]$, which essentially determine the part which is over 384-bit.
2. After addition, the sum is longer than 384-bit. We separated the sum into two parts. The first part is lower than 2^{379}, namely, $((T1[3]\&\texttt{0x07}) \parallel T1[0 \sim 2])$, we directly store them in memory (line 2).
3. The second part is higher than 2^{379}, namely, $(T1[3 \sim 5]) \gg 3$. we multiply it by $c2$, and store the product in three temporary registers $T2[0 \sim 2]$.
4. Finally, the remaining parts $(ZH[0 \sim 43])$ are multiplied by constant c and added to the intermediate results $(ZL[0 \sim 47])$ and reduction results $T2[0 \sim 2]$.

Algorithm 11 requires an execution time of 1018 clock cycles including the stack operation at the beginning and end of the Assembly function (e.g., `PUSH`s and `POP`s).

$2^{384} - 317$. Similar as Algorithm 11 for $2^{379} - 19$, the reduction of p_{384} needs an execution time of 1157 clock cycles. The fast reduction algorithm shown in Algorithm 12 can be described as follows:

1. We perform the first reduction (line 1), i.e. the computation of $(c', Z') = ZH \cdot c + ZL$. Instead of computing the whole $ZH \cdot c + ZL$, we only compute $T[0 \sim 5] = ZH[44 \sim 47] \cdot c + ZL[44 \sim 47]$, which essentially determine the part which is over 384-bit.
2. We separated the sum into two parts. The first part is lower than 2^{384}, namely, $(T1[0 \sim 3])$, we directly store them in memory (line 2).
3. The second part is higher than 2^{384}, namely, $T1[4 \sim 5]$. we multiply it by c, and store the product in three temporary registers $T2[0 \sim 3]$.

Algorithm 11 Fast (Incomplete) Reduction for p_{379}

```
Require: A $2n$-bit product $Z = ZH \cdot 2^{384} + ZL$, the constants $c1 = 2^5 \cdot 19$ and $c2 = 19$.
Ensure: The incomplete reduction result $R = Z \bmod p_1 \in [0, 2^{380}]$.
1: $T1[0 \sim 5] \leftarrow ZH[44 \sim 47] \times c1 + ZL[44 \sim 47]$
   {The first reduction is based on the equation: $2^{384} \equiv 608 \bmod p_1$}
2: $ZL[44 \sim 47] \leftarrow ((T1[3]\&\texttt{0x07}) \parallel T1[0 \sim 2])$
   {Get and store the bit-subsection, which is less than $2^{379}$}
3: $T2[0 \sim 2] \leftarrow (T1[3 \sim 5] \gg 3) \times 19$
   {The second reduction is based on the equation: $2^{379} \equiv 19 \bmod p_1$}.
4: $R[0 \sim 47] \leftarrow (ZH[0 \sim 43] \times c1) + ZL[0 \sim 47] + T2[0 \sim 2]$
   {The second reduction.}
5: return $R$
```

4. The remaining parts ($ZH[0 \sim 43]$) are multiplied by constant c and added to the intermediate results ($ZL[0 \sim 47]$) and reduction results $T2[0 \sim 3]$.
5. Finally, the *carry* bit is multiplied by constant c and added to results $R[0 \sim 47]$.

Algorithm 12 Fast (Incomplete) Reduction for p_{384}

```
Require: A $2n$-bit product $Z = ZH \cdot 2^{384} + ZL$, the constants $c = 317$.
Ensure: The incomplete reduction result $R = Z \bmod 2^{384} - 317 \in [0, 2^{384}]$.
1: $T1[0 \sim 5] \leftarrow ZH[44 \sim 47] \times c + ZL[44 \sim 47]$
2: $ZL[44 \sim 47] \leftarrow T1[0 \sim 3]$
3: $T2[0 \sim 3] \leftarrow T1[4 \sim 5] \times c$
4: $\{carry, R[0 \sim 47]\} \leftarrow (ZH[0 \sim 43] \times c) + ZL[0 \sim 47] + T2[0 \sim 3]$
5: $R[0 \sim 47] \leftarrow carry \times c + R[0 \sim 47]$
6: return $R$
```

3.2 Field Inversion

Constant-time inversion can be obtained by using Fermat's little theorem $a^{-1} = a^{p-2} \bmod p$.

The inversion of NUMS256 requires $255S + 12M$ as shown in Algorithm 13, it costs $1,218,645$ cycles. Similarly, the inversion operation of Ted379 requires an execution time of $378S + 12M$ as shown in Algorithm 14. On an 8-bit AVR processor, the inversion requires $3,566,477$ clock cycles.

Similarly, the Fermat-based inversion requires $383S + 13M$. The addition chain is given in Algorithm 15.

3.3 Addition and Subtraction

NUMS256 The implementation of addition and subtraction is similar as NUMS384.

Algorithm 13 Fermat-based inversion mod p_{256}

```
Require: Integer $A$ satisfying $1 \leq A \leq p-1$.
Ensure: Inverse $Z = A^{p-2} \bmod p = A^{-1} \bmod p$.
 1: $a_6 \leftarrow (a^2 \cdot a)^2$                                  {exp: 6, cost: 2S+1M}
 2: $t_1 \leftarrow ((a_6)^2)^2 \cdot a_6 \cdot a$                    {exp: $2^5-1$, cost: 2S+2M}
 3: $t_2 \leftarrow ((t_1)^2)^5 \cdot t_1$                            {exp: $2^{10}-1$, cost: 5S+1M}
 4: $t_3 \leftarrow ((a_6)^2)^2 \cdot a_6 \cdot a$                    {exp: $2^{20}-1$, cost: 10S+1M}
 5: $t_4 \leftarrow (((t_3)^2)^{10} \cdot t_2)^2 \cdot a$             {exp: $2^{31}-1$, cost: 11S+2M}
 6: $t_5 \leftarrow (((t_4)^2)^{31}) \cdot t_4$                       {exp: $2^{62}-1$, cost: 31S+1M}
 7: $t_6 \leftarrow (((t_5)^2)^{62}) \cdot t_5$                       {exp: $2^{124}-1$, cost: 62S+1M}
 8: $t_7 \leftarrow (((t_6)^2)^{124}) \cdot t_6$                      {exp: $2^{248}-1$, cost: 124S+1M}
 9: $Z \leftarrow (((((t_6)^2)^2 \cdot a)^2)^6) \cdot a$              {exp: $2^{256}-191$, cost: 8S+2M}
10: return $Z$
```

Algorithm 14 Fermat-based inversion mod p_{379}

```
Require: Integer $A$ satisfying $1 \leq A \leq p-1$.
Ensure: Inverse $Z = A^{p-2} \bmod p = A^{-1} \bmod p$.
 1: $a_2 \leftarrow a^2$                                 {exp: 2, cost: 1S+0M}
 2: $a_9 \leftarrow (a_2)^{2^2} \cdot a$                 {exp: 9, cost: 2S+1M}
 3: $a_{11} \leftarrow (a_9) \cdot a_2$                  {exp: 11, cost: 0S+1M}
 4: $t_1 \leftarrow (a_{11})^2 \cdot a_9$                {exp: $2^5-1$, cost: 1S+1M}
 5: $t_2 \leftarrow (t_1)^{2^5} \cdot t_1$               {exp: $2^{10}-1$, cost: 5S+1M}
 6: $t_3 \leftarrow (t_2)^{2^1} \cdot a$                 {exp: $2^{11}-1$, cost: 1S+1M}
 7: $t_4 \leftarrow (t_3)^{2^{11}} \cdot t_3$            {exp: $2^{22}-1$, cost: 11S+1M}
 8: $t_5 \leftarrow (t_4)^{2^{22}} \cdot t_4$            {exp: $2^{44}-1$, cost: 22S+1M}
 9: $t_6 \leftarrow (t_5)^{2^{44}} \cdot t_5$            {exp: $2^{88}-1$, cost: 44S+1M}
10: $t_7 \leftarrow (t_6)^{2^{88}} \cdot t_6$            {exp: $2^{176}-1$, cost: 88S+1M}
11: $t_8 \leftarrow (t_7)^{2^{176}} \cdot t_7$           {exp: $2^{352}-1$, cost: 176S+1M}
12: $t_9 \leftarrow (t_8)^{2^{22}} \cdot t_4$            {exp: $2^{374}-1$, cost: 22S+1M}
13: $Z \leftarrow (t_9)^{2^5} \cdot a_{11}$              {exp: $2^{379}-21$, cost: 5S+1M}
14: return $Z$
```

Ted379 In point arithmetic operation, it is easy to observe that the inputs of A and B are in the range of $[0, 2^{383}-1]$, thus, the sum of them is tolerated within s words, i.e. in the rang of $[0, 2^{384}-1]$. Our implementation simply employ the multi-precision addition for field addition.

In point arithmetic (e.g., point doubling), there exists the case the field subtraction is executed before the addition. In order to avoid the overflow of addition function, we keep the output of field subtraction within the range of $[0, 2^{383}-1]$. The concrete implementation is shown in Algorithm 17.

NUMS384 We implement both of field addition and subtraction in an incomplete fashion, both of them requires two round reduction. The modular addition and subtraction are given in Algorithm 18 and Algorithm 19, respectively.

Algorithm 15 Fermat-based inversion mod p_{384}

```
Require: Integer $A$ satisfying $1 \leq A \leq p-1$.
Ensure: Inverse $Z = A^{p-2} \bmod p = A^{-1} \bmod p$.
 1: $a_3 \leftarrow a^2 \cdot a$                                  $\{$exp: 3, cost: 1S+1M$\}$
 2: $a_{15} \leftarrow (a_3)^{2^2} \cdot a_3$                       $\{$exp: 15, cost: 2S+1M$\}$
 3: $t_0 \leftarrow a_{15}^2 \cdot a$                             $\{$exp: $2^5-1$, cost: 1S+1M$\}$
 4: $t_1 \leftarrow t_0^{2^5} \cdot t_0$                           $\{$exp: $2^{10}-1$, cost: 5S+1M$\}$
 5: $t_2 \leftarrow (t_1)^{2^5} \cdot t_0$                         $\{$exp: $2^{15}-1$, cost: 5S+1M$\}$
 6: $t_3 \leftarrow (t_2)^{2^{15}} \cdot t_2$                      $\{$exp: $2^{30}-1$, cost: 15S+1M$\}$
 7: $t_4 \leftarrow (t_3)^{2^{30}} \cdot t_3$                      $\{$exp: $2^{60}-1$, cost: 30S+1M$\}$
 8: $t_5 \leftarrow (t_4)^{2^{60}} \cdot t_4$                      $\{$exp: $2^{120}-1$, cost: 60S+1M$\}$
 9: $t_6 \leftarrow (t_5)^{2^{120}} \cdot t_5$                     $\{$exp: $2^{240}-1$, cost: 120S+1M$\}$
10: $t_7 \leftarrow (t_6)^{2^{120}} \cdot t_5$                     $\{$exp: $2^{360}-1$, cost: 120S+1M$\}$
11: $t_8 \leftarrow (t_7)^{2^{15}} \cdot t_2$                      $\{$exp: $2^{375}-1$, cost: 15S+1M$\}$
12: $t_9 \leftarrow ((t_8)^{2^3} \cdot a_3)^{2^6}) \cdot a$          $\{$exp: $2^{384}-319$, cost: 9S+2M$\}$
13: return $t_9$
```

Algorithm 16 Multi-precision addition for p_{379}

```
Require: Two 48-word operands $A$ and $B$ in $[0, 2^{383}-1]$.
Ensure: The incomplete reduction result $R = A + B \bmod p_1 \in [0, 2^{384}-1]$.
1: $\varepsilon = 0$
2: for $i = 0$ to 47 do
3:     $(\varepsilon, r_i) \leftarrow a_i + b_i + \varepsilon$
4: return $R$
```

3.4 Summary of Execution time for Field Arithmetic Operations

The execution time of field arithmetic of Ted379 and Curve384317 is summarized in Table 7.

CMUL: $16 \cdot 256$-bit for NUMS256, $16 \cdot 384$-bit for NUMS384, $24 \cdot 379$-bit for Ted379.

3.5 Estimate Field Arithmetic of Curve41417 and Ed448

For fair comparison, the multiplication and squaring are also implemented using Karatsuba method. The first level Karatsuba is implemented in C, while the 208-bit and 224-bit multiplication and squaring are estimated based on the Karatsuba techniques. Our estimated method is gives as follows:

In Hutter-Schwabe's Karatsuba multiplication (Assembly):

$$T_{mul192}(i.e.\ 2996cc) \approx [(T_{256}(i.e.4936cc) + T_{128}(i.e.1361\ cc))]/2 \cdot 0.95 = 2991\ cc. \tag{5}$$

Thus,

$$T_{mul208} \approx [T_{256}(i.e.4936cc) + T_{160}(i.e.2023cc)]/2 \cdot 0.95 = 3305cc \tag{6}$$

Algorithm 17 Field subtraction for p_{379}

```
Require: Two 48-word operands $A$ and $B$ in $[0, 2^{384} - 1]$.
Ensure: The incomplete reduction result $R = A + B \bmod p_1 \in [0, 2^{383} - 1]$.
 1: $\varepsilon = 0$
 2: for $i = 0$ to 47 do
 3:     $(\varepsilon, r_i) \leftarrow a_i - b_i - \varepsilon$
 4: $r_{47} = r_{47}$ & $0x7F$
 5: $mask \leftarrow -\varepsilon \bmod 2^{2w}$
 6: $[\varepsilon, (r_1, r_0)] \leftarrow (r_1, r_0) - [(2^4 \cdot 19)\&mask]$
 7: for $i = 2$ to 47 do
 8:     $(\varepsilon, r_i) \leftarrow r_i - \varepsilon$
 9: return $R$
```

Algorithm 18 Field addition for p_{384}

```
Require: Two 48-word operands $A$ and $B$ in $[0, 2^{384} - 1]$.
Ensure: The incomplete reduction result $R = A + B \bmod p_2 \in [0, 2^{384} - 1]$.
 1: $\varepsilon = 0$
 2: for $i = 0$ to 47 do
 3:     $(\varepsilon, r_i) \leftarrow a_i + b_i + \varepsilon$
 4: $mask \leftarrow -\varepsilon \bmod 2^{16}$
 5: $[\varepsilon, (r_1, r_0)] \leftarrow (r_1, r_0) + (317$ & $mask)$
 6: for $i = 2$ to 47 do
 7:     $(\varepsilon, r_i) \leftarrow r_i + \varepsilon$
 8: $mask \leftarrow -\varepsilon \bmod 2^{16}$
 9: $[\varepsilon, (r_1, r_0)] \leftarrow (r_1, r_0) + (317$ & $mask)$
10: for $i = 2$ to 47 do
11:     $(\varepsilon, r_i) \leftarrow r_i + \varepsilon$
12: return $R$
```

$$T_{mul224} \approx [T_{256}(i.e.4936cc) + T_{192}(i.e.2996cc)]/2 \cdot 0.95 = 3767cc \tag{7}$$

In Seo's implementation for Karatsuba squaring (Assembly)

$$T_{sqr192}(i.e.\ 1982cc) \approx [(T_{256}(i.e.3297cc) + T_{128}(i.e.982\ cc))]/2 \cdot 0.93 = 1989\ cc. \tag{8}$$

Thus,

$$T_{sqr208} \approx [T_{256}(i.e.3297cc) + T_{160}(i.e.1415cc)]/2 \cdot 0.93 = 2191cc \tag{9}$$

$$T_{sqr224} \approx [T_{256}(i.e.3297cc) + T_{192}(i.e.1982cc)]/2 \cdot 0.93 = 2454cc \tag{10}$$

Our 384-bit Karatsuba mul requires 11982 cycles, which is exactly 4 times of 192-bit Hutter's Karatsuba implementation. Thus,

$$T_{mul414} \approx T_{mul208} \cdot 4 = 13220cc \tag{11}$$

Algorithm 19 Field subtraction for p_{384}

```
Require: Two 48-word operands $A$ and $B$ in $[0, 2^{384} - 1]$.
Ensure: The incomplete reduction result $R = A - B \bmod p_2 \in [0, 2^{384} - 1]$.
  $\varepsilon = 0$
  for $i = 0$ to 47 do
    $(\varepsilon, r_i) \leftarrow a_i - b_i - \varepsilon$
  $mask \leftarrow -\varepsilon \bmod 2^{16}$
  $[\varepsilon, (r_1, r_0)] \leftarrow (r_1, r_0) - (317 \,\&\, mask)$
  for $i = 2$ to 47 do
    $(\varepsilon, r_i) \leftarrow r_i - \varepsilon$
  $mask \leftarrow -\varepsilon \bmod 2^{16}$
  $[\varepsilon, (r_1, r_0)] \leftarrow (r_1, r_0) - (317 \,\&\, mask)$
  for $i = 2$ to 47 do
    $(\varepsilon, r_i) \leftarrow r_i - \varepsilon$
  return $R$
```

Table 7: Execution time field arithmetic for Ted379 and Curve384317 (unroll and looped fashions)

Operation	ADD	SUB	MUL	SQR	INV	CMUL
p_{256}	550	550	6,301	4,489	1,218,645	830
p_{379} (U/L)	363	686	**12,971**/15,455	**9,081**/10,496	**3,566,477**/4,132,598	1424
p_{384} (U/L)	959	959	**13,113**/15,590	**9,254**/10,663	**3,715,866**/4,287,720	1227

$$T_{mul448} \approx T_{mul224} \cdot 4 = 15068cc \tag{12}$$

For squaring, our 384-bit Karatsuba sqr requires 8114 cycles, which is exactly 4.094 times of 192-bit Seo et al's Karatsuba implementation.

$$T_{sqr414} \approx T_{sqr208} \cdot 4.094 = 8970cc \tag{13}$$

$$T_{sqr448} \approx T_{sqr224} \cdot 4.094 = 10046cc \tag{14}$$

Reduction of Curve41417 is similar as Ted379, two levels reduction are required, thus, the reduction of Curve41417,

$$T_{red414} \approx 1018 \cdot (414/379) = 1112cc \tag{15}$$

Reduction of Ed448 is similar as NUMS384, thus, the reduction of Ed448,

$$T_{red448} \approx 1157 \cdot (448/384) = 1350cc \tag{16}$$

Curve41417 can be implemented in a similar way as Ted37919, Ed448 is similar as NUMS384. Thus, we can get easily estimate the addition and subtraction based on our implementation. As we implement the addition and subtraction in 32-bit fashion, thus, $fp414add = 363/12 \cdot 13 = 394cc$; $fp414sub = 744cc$. The estimated results of Curve41417 and Ed448 is given in Table 8. The Fermat-based inversion of Ed448 requires 445M + 12S.

Table 8: Execution time field arithmetic for Curve41417 and Ed448

Operation	ADD	SUB	MUL	SQR	INV
$Curve41417$	394	744	14,332	9,902	NA
$Ed448$	1,119	1,119	16,418	11,396	5,268,236

3.6 Execution Times of Scalar Multiplication

The execution times of Ted379, NUMS384, Curve41417 and Ed448 are given in Table 4. We used window method ($w = 5$) for twisted Edwards curve and

Table 9: Performance comparison of scalar multiplication between Ted379, NUMS384, Curve41417 and Ed448

Operation	NUMS256	Ted379	NUMS384	Curve41417	Ed448
TWE.WIN ($w = 5$)	15,807,767	45,350,543	47,096,411†	49,919,039†	59,421,773†
MON. *	15,125,232	44,245,500	46,482,328†	48,727,345†	58,619,275†

†: Estimated results.

*: **Constant multiplication** $(A + 2)/4$ **is used in point doubling**.

Montgomery ladder algorithm for Montgomery form. We test the current implementation of Montgomery ladder method for Ted379, it requires 44,245,500 clock cycles, our estimated result using magma is 43,788,576. The error is 1%. Thus, we give the reasonable estimated results for NUMS384, Curve41417 and Ed448 in Table 4.

4 ECC Implementation (P-521) over NEON

Multi-precision modular multiplication and squaring are performance-critical building blocks of Elliptic Curve Cryptography (ECC). Since the algorithm is a computation-intensive operation, it demands careful optimizations to achieve acceptable performance. Recently, an increasing number of embedded processors started to employ Single Instruction Multiple Data (SIMD) instructions to perform massive body of multimedia workloads.

In order to exploit the parallel computing power of SIMD instructions, traditional cryptography software needs to be rewritten into a vectorized format. The most well known approach is a reduced-radix representation for a better handling of the carry propagation [34]. The redundant representation reduces the number of active bits per register. Keeping the final result within remaining capacity of a register can avoid carry propagations. In [11], vector instructions on the CELL microprocessor are used to perform multiplication on operands represented with a radix of 2^{16}. At CHES 2012, Bernstein and Schwabe adopted the reduced radix and presented an efficient modular multiplication on specific ECC curves. Since the target curves only have low hamming weight in the least significant bits, modular arithmetics are efficiently computed with multiplication and addition operations. At HPEC 2013, a multiplicand reduction method in the reduced-radix representation was introduced for the NIST curves [61]. However, the reduced-radix representation requires to compute more number of partial products than the non-redundant representation, because it needs more number of word to store previous radix 2^{32} variables into smaller radix. At CHES 2014, the ECC implementation adopts 2-level Karatsuba multiplication in the redundant representation [7]. Recently efficient Karatsuba multiplication algorithm for P-521 by [27] is proposed at PKC'15. However, this is designed for 64-bit SISD architecture not for ARM-NEON SIMD platforms. Until now, there is relatively few studied on NIST's (and SECG's) curve P-521 for NEON architecture. Since the curve is NIST standard and ARM-NEON is the most well known smart phone processor, the efficient ARM-NEON implementation of P-521 should be considered in high priority. In this section, we present speed record of P-521 over ARM-NEON platform. We exploit 1-level Karatsuba method in order to provide asymtotically faster integer multiplication and multiplication and accumulation operation for efficient reduction algorithm.

4.1 NIST curve P-521

The Weierstrass form NIST curve P-521 as standardised in [22, 58] and the finite field $\mathbb{F}_p$ is defined by:

$$p = 2^{521} - 1$$

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

$$\begin{aligned} a = \texttt{01FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF} \\ \texttt{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF} \\ \texttt{FFFFFFFF FFFFFFFF FFFFFFFC} \end{aligned}$$

$$\begin{aligned} b = \texttt{0051 953EB961 8E1C9A1F 929A21A0 B68540EE A2DA725B 99B315F3} \\ \texttt{B8B48991 8EF109E1 56193951 EC7E937B 1652C0BD 3BB1BF07 3573DF88} \\ \texttt{3D2C34F1 EF451FD4 6B503F00} \end{aligned}$$

and group order is defined by:

$n =$ `01FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFA 51868783 BF2F966B 7FCC0148 F709A5D0 3BB5C9B8 899C47AE BB6FB71E 91386409`

Using Jacobian projective coordinates, for $P_1 = (X_1, Y_1, Z_1)$ the point $2P_1 = (X_3, Y_3, Z_3)$ is computed as follows:

$$T_1 \leftarrow Z_1^2,\ T_2 \leftarrow Y_1^2,\ T_3 \leftarrow X_1 \cdot T_2,\ T_4 \leftarrow X_1 + T_1,\ T_5 \leftarrow X_1 - T_1,$$
$$T_6 \leftarrow T_4 \cdot T_5,\ T_4 \leftarrow 3 \cdot T_6,\ T_5 \leftarrow T_4^2,\ T_6 \leftarrow 8 \cdot T_3,\ X_3 \leftarrow T_5 - T_6,$$
$$T_5 \leftarrow Y_1 + Z_1,\ T_6 \leftarrow T_5^2,\ T_5 \leftarrow T_6 - T_1,\ Z_3 \leftarrow T_5 - T_2,\ T_5 \leftarrow 4 \cdot T_3,$$
$$T_6 \leftarrow T_5 - X_3,\ T_5 \leftarrow T_4 \cdot T_6,\ T_6 \leftarrow T_2^2,\ T_4 \leftarrow 8 \cdot T_6,\ Y_3 \leftarrow T_5 - T_4$$

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 as follows:

$$T_1 \leftarrow Z_1^2,\ T_2 \leftarrow T_1 \cdot Z_1,\ T_1 \leftarrow T_1 \cdot X_2,\ T_2 \leftarrow T_2 \cdot Y2,\ T_1 \leftarrow T_1 - X_1$$
$$T_2 \leftarrow T_2 - Y_1,\ Z_3 \leftarrow Z_1 \cdot T_1,\ T_3 \leftarrow T_1^2,\ T_4 \leftarrow T_3 \cdot T_1,\ T_3 \leftarrow T_3 \cdot X_1$$
$$T_1 \leftarrow 2 \cdot T_3,\ X_3 \leftarrow T_2^2,\ X_3 \leftarrow X_3 - T_1,\ X_3 \leftarrow X_3 - T_4,\ T_3 \leftarrow T_3 - X_3$$
$$T_3 \leftarrow T_3 \cdot T_2,\ T_4 \leftarrow T_4 \cdot Y_1,\ Y_3 \leftarrow T_3 - T_4$$

For a point P_2 which is projective 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 as follows:

$$T_1 \leftarrow Z_2^2,\ U_1 \leftarrow X_1 \cdot T_1,\ T_2 \leftarrow Z_1^2,\ U_2 \leftarrow X_2 \cdot T_2,\ T_3 \leftarrow Y_1 \cdot Z_2$$
$$S_1 \leftarrow T_3 \cdot T_1,\ T_4 \leftarrow Y_2 \cdot Z_1,\ S_2 \leftarrow T_4 \cdot T_2,\ H \leftarrow U_2 - U_1,\ R \leftarrow S_2 - S_1$$
$$T_1 \leftarrow R^2,\ T_2 \leftarrow H^2,\ T_3 \leftarrow T_2 \cdot H,\ T_4 \leftarrow U_1 \cdot T_2,\ T_1 \leftarrow T_1 - T_3$$
$$T_2 \leftarrow 2 \cdot T_4,\ X_3 \leftarrow T_1 - T_2,\ T_3 \leftarrow S_1 \cdot T_3 T_4 \leftarrow T_4 - X_3,\ T_4 \leftarrow R \cdot T_4$$
$$Y_3 \leftarrow T_4 - T_3,\ T_1 \leftarrow Z_1 \cdot Z_2,\ Z_3 \leftarrow H \cdot T_1$$

4.2 Proposed Method

Multiplication The prime of P-521 curve is $2^{521} - 1$. This representation can be written in $2^{522} - 2$ by following OpenSSL 1.0.0e approach. We choose 27/26-radix and this divides 522-bit into 20-limb as follows: (27, 26, 26, 26, 26, 26, 26, 26, 26, 26 || 27, 26, 26, 26, 26, 26, 26, 26, 26, 26). We applied 1 level of Karatsuba multiplication as described in Algorithm 20. Since the openrand length is 522-bit, we divide the operand into two 261-bit wise opernads. One level of Karatsuba multiplication reduces the multiplication complexity from one 522-bit multiplication to three 261-bit multiplication operations.

For 261-bit multiplication of operand in (27, 26, 26, 26, 26, 26, 26, 26, 26, 26) representation, we can conduct multiplication as follows. The equation shows

Algorithm 20 Karatsuba-based multiplication mod p_{521}

```
Require: Integer $a, b$ satisfying $1 \leq a, b \leq p - 1$.
Ensure: Results $z = a \cdot b \bmod p$.
 1: $a_L \leftarrow a\ mod\ 2^{261}$
 2: $a_H \leftarrow a\ div\ 2^{261}$
 3: $b_L \leftarrow b\ mod\ 2^{261}$
 4: $b_H \leftarrow b\ div\ 2^{261}$
 5: $r_L \leftarrow a_L \cdot b_L$
 6: $t \leftarrow (r_L - a_H \cdot b_H \cdot 2^{261})\ mod\ p$
 7: $t_H \leftarrow t\ div\ 2^{261}$
 8: $t_L \leftarrow t\ mod\ 2^{261}$
 9: $t_{HL} \leftarrow t_H - t_L$
10: $a_K \leftarrow a_L + a_H$
11: $b_K \leftarrow b_L + b_H$
12: $ab_K \leftarrow (t_{HL} + t_L - t_H \cdot 2^{261} + a_K \cdot b_K \cdot 2^{261})\ mod\ p$
13: return $ab_K$
```

that some of the partial product needs doubling the intermediate results to get proper results.

$$\begin{aligned}
c_0 &\leftarrow a_0b_0\\
c_1 &\leftarrow a_0b_1 + a_1b_0\\
c_2 &\leftarrow a_0b_2 + a_2b_0 + 2a_1b_1\\
c_3 &\leftarrow a_0b_3 + a_3b_0 + 2(a_1b_2 + a_2b_1)\\
c_4 &\leftarrow a_0b_4 + a_4b_0 + 2(a_1b_3 + a_3b_1 + a_2b_2)\\
c_5 &\leftarrow a_0b_5 + a_5b_0 + 2(a_1b_4 + a_4b_1 + a_2b_3 + a_3b_2)\\
c_6 &\leftarrow a_0b_6 + a_6b_0 + 2(a_1b_5 + a_5b_1 + a_2b_4 + a_4b_2 + a_3b_3)\\
c_7 &\leftarrow a_0b_7 + a_7b_0 + 2(a_1b_6 + a_6b_1 + a_2b_5 + a_5b_2 + a_3b_4 + a_4b_3)\\
c_8 &\leftarrow a_0b_8 + a_8b_0 + 2(a_1b_7 + a_7b_1 + a_2b_6 + a_6b_2 + a_3b_5 + a_5b_3 + a_4b_4)\\
c_9 &\leftarrow a_0b_9 + a_9b_0 + 2(a_1b_8 + a_8b_1 + a_2b_7 + a_7b_2 + a_3b_6 + a_6b_3 + a_4b_5 + a_5b_4)\\
c_{10} &\leftarrow 2(a_1b_9 + a_9b_1 + a_2b_8 + a_8b_2 + a_3b_7 + a_7b_3 + a_4b_6 + a_6b_4 + a_5b_5)\\
c_{11} &\leftarrow a_2b_9 + a_9b_2 + a_3b_8 + a_8b_3 + a_4b_7 + a_7b_4 + a_5b_6 + a_6b_5\\
c_{12} &\leftarrow a_3b_9 + a_9b_3 + a_4b_8 + a_8b_4 + a_5b_7 + a_7b_5 + a_6b_6\\
c_{13} &\leftarrow a_4b_9 + a_9b_4 + a_5b_8 + a_8b_5 + a_6b_7 + a_7b_6\\
c_{14} &\leftarrow a_5b_9 + a_9b_5 + a_6b_8 + a_8b_6 + a_7b_7\\
c_{15} &\leftarrow a_6b_9 + a_9b_6 + a_7b_8 + a_8b_7\\
c_{16} &\leftarrow a_7b_9 + a_9b_7 + a_8b_8\\
c_{17} &\leftarrow a_8b_9 + a_9b_8\\
c_{18} &\leftarrow a_9b_9
\end{aligned}$$

The NEON architecture supports 2-way 32-bit wise multiplication. In order to ensure high performance, we group the two adjcent partial products as follows. Firstly we output the intermediate results from c_0 but from second round we output the results from c_1. In order to align the results we conduct word-wise left shift operation.

$$
\begin{aligned}
(c_1, c_0) &\leftarrow (a_0b_1, a_0b_0)\\
(c_3, c_2) &\leftarrow (a_0b_3, a_0b_2) + (a_3b_0, a_2b_0)\\
(c_5, c_4) &\leftarrow (a_0b_5, a_0b_4) + 2(a_2b_3, a_2b_2) + (a_5b_0, a_4b_0)\\
(c_7, c_6) &\leftarrow (a_0b_7, a_0b_6) + 2(a_2b_5, a_2b_4) + 2(a_4b_3, a_4b_2) + (a_7b_0, a_6b_0)\\
(c_9, c_8) &\leftarrow (a_0b_9, a_0b_8) + 2(a_2b_7, a_2b_6) + 2(a_4b_5, a_4b_4) + 2(a_6b_3, a_6b_2) + (a_9b_0, a_8b_0)\\
(c_{11}, c_{10}) &\leftarrow (a_2b_9, 2a_2b_8) + (a_4b_7, 2a_4b_6) + (a_6b_5, 2a_6b_4) + (a_8b_3, 2a_8b_2)\\
(c_{13}, c_{12}) &\leftarrow (a_4b_9, a_4b_8) + (a_6b_7, a_6b_6) + (a_8b_5, a_8b_4)\\
(c_{15}, c_{14}) &\leftarrow (a_6b_9, a_6b_8) + (a_8b_7, a_8b_6)\\
(c_{17}, c_{16}) &\leftarrow (a_8b_9, a_8b_8)\\
c &\leftarrow c \ll word\\
(c_2, c_1) &\leftarrow (2a_1b_1, a_1b_0)\\
(c_4, c_3) &\leftarrow 2(a_1b_3, a_1b_2) + 2(a_3b_1, a_2b_1)\\
(c_6, c_5) &\leftarrow 2(a_1b_5, a_1b_4) + 2(a_3b_3, a_3b_2) + 2(a_5b_1, a_4b_1)\\
(c_8, c_7) &\leftarrow 2(a_1b_7, a_1b_6) + 2(a_3b_5, a_3b_4) + 2(a_5b_3, a_5b_2) + 2(a_7b_1, a_6b_1)\\
(c_{10}, c_9) &\leftarrow 2(a_1b_9, a_1b_8) + 2(a_3b_7, a_3b_6) + 2(a_5b_5, a_5b_4) + 2(a_7b_3, a_7b_2) + 2(a_9b_1, a_8b_1)\\
(c_{12}, c_{11}) &\leftarrow (a_3b_9, a_3b_8) + (a_5b_7, a_5b_6) + (a_7b_5, a_7b_4) + (a_9b_3, a_9b_2)\\
(c_{14}, c_{13}) &\leftarrow (a_5b_9, a_5b_8) + (a_7b_7, a_7b_6) + (a_9b_5, a_9b_4)\\
(c_{16}, c_{15}) &\leftarrow (a_7b_9, a_7b_8) + (a_9b_7, a_9b_6)\\
(c_{18}, c_{17}) &\leftarrow (a_9b_9, a_9b_8)
\end{aligned}
$$

Step 6 in Algorithm 20 conducts the partial product of $a_H \cdot b_H$ together with modular reduction. From c_{11} to c_{19}, the part of 261-bit multiplication is conducted. The c_{20} is multiplied and subtracted the results. From c_{21} to c_{29}, the partial products are directly subtracted to intermediate results.

$$
\begin{aligned}
(a_9 \sim a_0) &\leftarrow (a_{19} \sim a_{10}) \\
(b_9 \sim b_0) &\leftarrow (b_{19} \sim b_{10}) \\
(c_{12}, c_{11}) &\leftarrow (c_{12}, c_{11}) + (2a_1b_1, a_1b_0) \\
(c_{14}, c_{13}) &\leftarrow (c_{14}, c_{13}) + 2(a_1b_3, a_1b_2) + 2(a_3b_1, a_2b_1) \\
(c_{16}, c_{15}) &\leftarrow (c_{16}, c_{15}) + 2(a_1b_5, a_1b_4) + 2(a_3b_3, a_3b_2) + 2(a_5b_1, a_4b_1) \\
(c_{18}, c_{17}) &\leftarrow (c_{18}, c_{17}) + 2(a_1b_7, a_1b_6) + 2(a_3b_5, a_3b_4) + 2(a_5b_3, a_5b_2) + 2(a_7b_1, a_6b_1) \\
(t_1, t_0) &\leftarrow (4a_1b_9, 2a_1b_8) + (4a_3b_7, 2a_3b_6) + (4a_5b_5, 2a_5b_4) + \\
&\quad (4a_7b_3, 2a_7b_2) + (4a_9b_1, 2a_8b_1) \\
c_{19} &\leftarrow c_{19} + t_0 \\
c_0 &\leftarrow c_0 - t_1 \\
(c_2, c_1) &\leftarrow (c_2, c_1) - 2(a_3b_9, a_3b_8) - 2(a_5b_7, a_5b_6) - 2(a_7b_5, a_7b_4) - 2(a_9b_3, a_9b_2) \\
(c_4, c_3) &\leftarrow (c_4, c_3) - 2(a_5b_9, a_5b_8) - 2(a_7b_7, a_7b_6) - 2(a_9b_5, a_9b_4) \\
(c_6, c_5) &\leftarrow (c_6, c_5) - 2(a_7b_9, a_7b_8) - 2(a_9b_7, a_9b_6) \\
(c_8, c_7) &\leftarrow (c_8, c_7) - 2(a_9b_9, a_9b_8) \\
c &\leftarrow c \gg word \\
(c_{11}, c_{10}) &\leftarrow (c_{11}, c_{10}) + (a_0b_1, a_0b_0) \\
(c_{13}, c_{12}) &\leftarrow (c_{13}, c_{12}) + (a_0b_3, a_0b_2) + (a_3b_0, a_2b_0) \\
(c_{15}, c_{14}) &\leftarrow (c_{15}, c_{14}) + (a_0b_5, a_0b_4) + 2(a_2b_3, a_2b_2) + (a_5b_0, a_4b_0) \\
(c_{17}, c_{16}) &\leftarrow (c_{17}, c_{16}) + (a_0b_7, a_0b_6) + 2(a_2b_5, a_2b_4) + 2(a_4b_3, a_4b_2) + (a_7b_0, a_6b_0) \\
(c_{19}, c_{18}) &\leftarrow (c_{19}, c_{18}) + (a_0b_9, a_0b_8) + 2(a_2b_7, a_2b_6) + \\
&\quad 2(a_4b_5, a_4b_4) + 2(a_6b_3, a_6b_2) + (a_9b_0, a_8b_0) \\
(t_1, t_0) &\leftarrow (2a_2b_9, 4a_2b_8) + (2a_4b_7, 4a_4b_6) + (2a_6b_5, 4a_6b_4) + (2a_8b_3, 4a_8b_2) \\
c_0 &\leftarrow c_0 - t_0 \\
c_1 &\leftarrow c_1 - t_1 \\
(c_3, c_2) &\leftarrow (c_3, c_2) - 2(a_4b_9, a_4b_8) - 2(a_6b_7, a_6b_6) - 2(a_8b_5, a_8b_4) \\
(c_5, c_4) &\leftarrow (c_5, c_4) - 2(a_6b_9, a_6b_8) - 2(a_8b_7, a_8b_6) \\
(c_7, c_6) &\leftarrow (c_7, c_6) - 2(a_8b_9, a_8b_8)
\end{aligned}
$$

Squaring Multi-precision squaring can be utilized with ordinary multiplication methods. However, squaring dedicated method has two advantages over the multiplication methods for squaring computations. Both partial products $A[i] \times A[j]$ and $A[j] \times A[i]$ output the same results. By taking accounts of the feature, the parts are multiplied with doubled form (i.e. $2 \times A[i] \times A[j]$) which provides the same results of conventional multiplication (i.e. $A[i] \times A[j] + A[j] \times A[i]$). We applied squaring on 261-bit wise operand as follows.

$$\begin{aligned}
c_0 &\leftarrow a_0a_0\\
c_1 &\leftarrow 2(a_0a_1)\\
c_2 &\leftarrow 2(a_0a_2 + a_1a_1)\\
c_3 &\leftarrow 2(a_0a_3) + 4(a_1a_2)\\
c_4 &\leftarrow 2(a_0a_4 + a_2a_2) + 4(a_1a_3)\\
c_5 &\leftarrow 2(a_0a_5) + 4(a_1a_4 + a_2a_3)\\
c_6 &\leftarrow 2(a_0a_6 + a_3a_3) + 4(a_1a_5 + a_2a_4)\\
c_7 &\leftarrow 2(a_0a_7) + 4(a_1a_6 + a_2a_5 + a_3a_4)\\
c_8 &\leftarrow 2(a_0a_8 + a_4a_4) + 4(a_1a_7 + a_2a_6 + a_3a_5)\\
c_9 &\leftarrow 2(a_0a_9) + 4(a_1a_8 + a_2a_7 + a_3a_6 + a_4a_5)\\
c_{10} &\leftarrow 2(a_5a_5) + 4(a_1a_9 + a_2a_8 + a_3a_7 + a_4a_6)\\
c_{11} &\leftarrow 2(a_2a_9 + a_3a_8 + a_4a_7 + a_5a_6)\\
c_{12} &\leftarrow 2(a_3a_9 + a_4a_8 + a_5a_7) + a_6a_6\\
c_{13} &\leftarrow 2(a_4a_9 + a_5a_8 + a_6a_7)\\
c_{14} &\leftarrow 2(a_5a_9 + a_6a_8) + a_7a_7\\
c_{15} &\leftarrow 2(a_6a_9 + a_7a_8)\\
c_{16} &\leftarrow 2(a_7a_9) + a_8a_8\\
c_{17} &\leftarrow 2(a_8a_9)\\
c_{18} &\leftarrow a_9a_9
\end{aligned}$$

For SIMD architecture, we group the two partial prodcuts as follows.

$$
\begin{aligned}
(c_1, c_0) &\leftarrow (2a_0a_1, a_0a_0)\\
(c_3, c_2) &\leftarrow 2(a_0a_3, a_0a_2) + (4a_1a_2, 2a_1a_1)\\
(c_5, c_4) &\leftarrow 2(a_0a_5, a_0a_4) + 4(a_1a_4, a_1a_3) + (4a_2a_3, 2a_2a_2)\\
(c_7, c_6) &\leftarrow 2(a_0a_7, a_0a_6) + 4(a_1a_6, a_1a_5) + 4(a_2a_5, a_2a_4) + (4a_3a_4, 2a_3a_3)\\
(c_9, c_8) &\leftarrow 2(a_0a_9, a_0a_8) + 4(a_1a_8, a_1a_7) + 4(a_2a_7, a_2a_6) +\\
&\quad 4(a_3a_6, a_3a_5) + (4a_4a_5, 2a_4a_4)\\
(c_{11}, c_{10}) &\leftarrow (2a_2a_9, 4a_1a_9) + (2a_3a_8, 4a_2a_8) + (2a_4a_7, 4a_3a_7) + (2a_5a_6, 4a_4a_6)\\
(c_{13}, c_{12}) &\leftarrow 2(a_4a_9, a_3a_9) + 2(a_5a_8, a_4a_8) + 2(a_6a_7, a_5a_7)\\
(c_{15}, c_{14}) &\leftarrow 2(a_6a_9, a_5a_9) + 2(a_7a_8, a_6a_8)\\
(c_{17}, c_{16}) &\leftarrow 2(a_8a_9, a_7a_9)\\
(t_1, t_0) &\leftarrow (a_6a_6, 2a_5a_5)\\
(t_3, t_2) &\leftarrow (a_8a_8, a_7a_7)\\
t_4 &\leftarrow a_9a_9\\
c_{10} &\leftarrow c_{10} + t_0\\
c_{12} &\leftarrow c_{12} + t_1\\
c_{14} &\leftarrow c_{14} + t_2\\
c_{16} &\leftarrow c_{16} + t_3\\
c_{18} &\leftarrow c_{18} + t_4
\end{aligned}
$$

We applied Step 6 in Algorithm 20 for the squaring method as well. Firstly we conduct the partial product of $a_H \cdot a_H$ together with modular reduction. From c_{11} to c_{19}, the part of 261-bit multiplication is conducted. The c_{20} is multiplied and subtracted the results. From c_{21} to c_{29}, the partial products are directly subtracted to intermediate results.

$$\begin{aligned}
(a_9 \sim a_0) &\leftarrow (a_{19} \sim a_{10}) \\
(c_{11}, c_{10}) &\leftarrow (c_{11}, c_{10}) + (2a_0a_1, a_0a_0) \\
(c_{13}, c_{12}) &\leftarrow (c_{13}, c_{12}) + 2(a_0a_3, a_0a_2) + (4a_1a_2, 2a_1a_1) \\
(c_{15}, c_{14}) &\leftarrow (c_{15}, c_{14}) + 2(a_0a_5, a_0a_4) + 4(a_1a_4, a_1a_3) + (4a_2a_3, 2a_2a_2) \\
(c_{17}, c_{16}) &\leftarrow (c_{17}, c_{16}) + 2(a_0a_7, a_0a_6) + 4(a_1a_6, a_1a_5) + 4(a_2a_5, a_2a_4) + (4a_3a_4, 2a_3a_3) \\
(c_{19}, c_{18}) &\leftarrow (c_{19}, c_{18}) + 2(a_0a_9, a_0a_8) + 4(a_1a_8, a_1a_7) + \\
&\quad 4(a_2a_7, a_2a_6) + 4(a_3a_6, a_3a_5) + (4a_4a_5, 2a_4a_4) \\
(c_1, c_0) &\leftarrow (c_1, c_0) - (4a_2a_9, 8a_1a_9) + (4a_3a_8, 8a_2a_8) + (4a_4a_7, 8a_3a_7) + (4a_5a_6, 8a_4a_6) \\
(c_3, c_2) &\leftarrow (c_3, c_2) - 4(a_4a_9, a_3a_9) - 4(a_5a_8, a_4a_8) - 4(a_6a_7, a_5a_7) \\
(c_5, c_4) &\leftarrow (c_5, c_4) - 4(a_6a_9, a_5a_9) - 4(a_7a_8, a_6a_8) \\
(c_7, c_6) &\leftarrow (c_7, c_6) - 4(a_8a_9, a_7a_9) \\
(t_1, t_0) &\leftarrow (2a_6a_6, 4a_5a_5) \\
(t_3, t_2) &\leftarrow 2(a_8a_8, a_7a_7) \\
t_4 &\leftarrow 2a_9a_9 \\
c_0 &\leftarrow c_{10} - t_0 \\
c_2 &\leftarrow c_{12} - t_1 \\
c_4 &\leftarrow c_{14} - t_2 \\
c_6 &\leftarrow c_{16} - t_3 \\
c_8 &\leftarrow c_{18} - t_4
\end{aligned}$$

Inversion Constant-time inversion is performed by powering by $p_{521} - 2 = 2^{521} - 3$ Then the inverse of x can be computed at a cost of 520S + 13M, as follows:

Addition and Subtraction Addition and subtraction over redundant representation do not introduce the carry or borrow propagation from least significant word to most significant word. We conduct 20 26/27-radix addition/subtraction with five times of 32-bit wise vector addition and subtraction operations. For point addition and doubling, addition variants such as double, triple, quadruple, octuple are needed. We also exploit vector addition 1, 2, 2, 3 times for double, triple, quadruple and octuple operations.

Reduction The multiplication and squaring computations produces a product of the 63-bit 20 limbs. We then use a sequence of carries to bring each limb down to 26 or 27 bits. We vectorize between a carry $c_0 \rightarrow c_1$ and $c_{10} \rightarrow c_{11}$, between a carry $c_1 \rightarrow c_2$ and $c_{11} \rightarrow c_{12}$. The computation order is as follows: $(c_{11}, c_1) \rightarrow (c_{10}, c_0)$, $(c_{12}, c_2) \rightarrow (c_{11}, c_1)$, $(c_{13}, c_3) \rightarrow (c_{12}, c_2)$, $(c_{14}, c_4) \rightarrow (c_{13}, c_3)$, $(c_{15}, c_5) \rightarrow (c_{14}, c_4)$, $(c_{16}, c_6) \rightarrow (c_{15}, c_5)$, $(c_{17}, c_7) \rightarrow (c_{16}, c_6)$, $(c_{18}, c_8) \rightarrow (c_{17}, c_7)$, $(c_{19}, c_9) \rightarrow (c_{18}, c_8)$, $(c_0, c_{10}) \rightarrow (c_{19}, c_9)$, $(c_{11}, c_1) \rightarrow (c_{10}, c_0)$. The computations output 20 limbs of results (27, 27, 26, 26, 26, 26, 26, 26, 26, 26 || 27, 27, 26, 26, 26, 26, 26, 26, 26, 26).

Algorithm 21 Fermat-based inversion mod p_{521}

```
Require: Integer $a_1$ satisfying $1 \le a_1 \le p-1$.
Ensure: Inverse $z = a_1^{p-2} \bmod p = a_1^{-1} \bmod p$.
 1: $a_2 \leftarrow a_1^2 \cdot a_1$                                {cost: 1S+1M}
 2: $a_3 \leftarrow a_2^2 \cdot a_1$                                {cost: 1S+1M}
 3: $a_6 \leftarrow a_3^{2^3} \cdot a_3$                            {cost: 3S+1M}
 4: $a_7 \leftarrow a_6^2 \cdot a_1$                                {cost: 1S+1M}
 5: $a_8 \leftarrow a_7^2 \cdot a_1$                                {cost: 1S+1M}
 6: $a_{16} \leftarrow a_8^{2^8} \cdot a_8$                         {cost: 8S+1M}
 7: $a_{32} \leftarrow a_{16}^{2^{16}} \cdot a_{16}$                {cost: 16S+1M}
 8: $a_{64} \leftarrow a_{32}^{2^{32}} \cdot a_{32}$                {cost: 32S+1M}
 9: $a_{128} \leftarrow a_{64}^{2^{64}} \cdot a_{64}$               {cost: 64S+1M}
10: $a_{256} \leftarrow a_{128}^{2^{128}} \cdot a_{128}$            {cost: 128S+1M}
11: $a_{512} \leftarrow a_{256}^{2^{256}} \cdot a_{256}$            {cost: 256S+1M}
12: $a_{519} \leftarrow a_{512}^{2^7} \cdot a_7$                    {cost: 7S+1M}
13: $a_1^{2^{521}-3} \leftarrow a_{519}^{2^2} \cdot a_1$            {cost: 2S+1M}
14: return $a_1^{2^{521}-3}$
```

The addition and subtraction computations carry out the 31-bit wise 20 limbs. Similarly, we use a sequence of carries to bring each limb down to 26 or 27 bits. The computation order is as follows: $(c_0, c_{10}) \rightarrow (c_{19}, c_9)$, $(c_{11}, c_1) \rightarrow (c_{10}, c_0)$, $(c_{12}, c_2) \rightarrow (c_{11}, c_1)$, $(c_{13}, c_3) \rightarrow (c_{12}, c_2)$, $(c_{14}, c_4) \rightarrow (c_{13}, c_3)$, $(c_{15}, c_5) \rightarrow (c_{14}, c_4)$, $(c_{16}, c_6) \rightarrow (c_{15}, c_5)$, $(c_{17}, c_7) \rightarrow (c_{16}, c_6)$, $(c_{18}, c_8) \rightarrow (c_{17}, c_7)$, $(c_{19}, c_9) \rightarrow (c_{18}, c_8)$. This computation outputs 20 limbs as follows (27, 26, 26, 26, 26, 26, 26, 26, 26, 27 || 27, 26, 26, 26, 26, 26, 26, 26, 26, 27).

Scalar Multiplication For unknown point we conduct the window method. This consists of pre-computation of point and window wise scalar multiplication. We tested over three different window sizes including 4, 5 and 6. For window size 4, pre-computation needs 1 time of doubling and 7 times of addition. For window size 5, pre-computation needs 1 time of doubling and 15 times of addition. For window size 6, pre-computation needs 1 time of doubling and 31 times of addition. By using pre-computed point, scalar multiplication needs (131A+520D), (105A+520D), (87A+516D) for 4, 5, 6 window methods. Total (138A+521D), (120A+521D), (118A+517D) are needed for 4, 5, 6 window methods. For fixed point we conduct the comb window method. Fixed point does not need pre-computation. Total (131A+130D), (105A+104D), (87A+86D) are needed for 4, 5, 6 window methods.

4.3 Evaluation

Table 10 shows the results of ECC over openssl. As the length of operand increase, the required clock cycle also increases. For P-521, 23.8/18.7M clock cycles are needed for A9 and A15 processors.

Table 10: Prime-field ECC timings from `openssl speed ecdh` on Cortex-A9 and Cortex-A15 devices where Cortex-A9 with linux and OpenSSL 1.0.2d on a Odroid-X2 development board running at 1.7 GHz and Cortex-A15 with linux and OpenSSL 1.0.2d on a Odroid-XU development board running at 1.6 GHz

Curve	A9 op/s	cycles	A15 op/s	cycles
`secp160r1`	1014.4	1,700,000	1258.8	1,280,000
`nist192`	718.2	2,380,000	951.0	1,760,000
`nist224`	489.2	3,400,000	701.4	2,240,000
`nist256`	475.5	3,570,000	574.9	2,720,000
`nist384`	154.6	11,050,000	223.0	7,200,000
`nist521`	71.2	23,800,000	85.3	18,720,000

In Table 14, the detailed clock cycles for basic operations are drawn. The clock cycles of finite field operations include reduction overheads. The performance of point addition and doubling is jacobian representation.

Table 11: Clock cycles for finite field multiplication, squaring and inversion; point addition and doubling

Target	Mul	Sqr	Inv	Point Add	Point Dbl
Cortex-A9	708	578	311451	12453	8036
Cortex-A15	350	276	149208	6176	3962

There are several works done over lower security levels including Curve41417 and Ed448-Goldilocks [31, 7]. However it is hard to retireve the fair performance evaluations due to diffierent features and parameters of target curve. For this reason, we set the obvious candidate regarding comparisons with previous benchmarks. We test openSSL implementations using the command openssl speed ecdh. On the same architecture, version 1.0.2d reports 71.2 and 85.3 operations per second for A9 and A15, which implies a count of approximately 23.8M and 18.7M cycles per ECDH.

Table 12: Clock cycles for scalar multiplication

Target	Unknown w=4	w=5	w=6	Fixed w=4	w=5	w=6	ECDH
Cortex-A9	6291936	6098946	6011768	3056410	2527714	2147404	8159172
Cortex-A15	3097904	3003728	2970976	1503661	1243027	1056902	4027878

5 ECC Implementation (K-571) over NEON

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 [49, 59, 74, 68, 71]. Recently, many modern embedded processors adopt the advanced built-in binary field multiplication. ARMv7 supports `VMULL.P8` operation which can compute eight 8-bit wise polynomial multiplications with single instruction. In [13], 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 [26], 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 section, 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.

5.1 Related Works

Koblitz curve $\mathbb{F}_{2^{571}}$ The 571-bit Koblitz elliptic curve namely K-571 standardized in [22] 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:

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

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]:

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

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.

ARM Processor ARM processor is a well known family of RISC processor architectures introduced in 1985 [76]. 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 [26].

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 [36]. 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 17.

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

Karatsuba method roughly executes $\frac{3s^2}{4}$ `mul` instructions to multiply two s-word operands [28]. Recently, the refined Karatsuba's algorithm from a Crypto 2009 paper by Bernstein [6] makes efficient use of the available registers to keep the low overheads from `load` and `store` instructions.

5.2 Proposed Method

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 22 and 33, respectively.

Algorithm 22 571-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:     $C[i] = A[i] \oplus A[i]$
```

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, straight-forward 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 13. 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 23. 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 13: Comparison of 192-bit polynomial multiplication methods

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

Algorithm 23 192-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 $C$}
```

After then we conduct the 571-bit multiplication with the 192-bit wise multiplication (refer Algorithm 23). 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 [36]. 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 17.

$$\begin{aligned}(A_H \cdot B_H \cdot 2^{\frac{2s}{3}} + A_M \cdot B_M \cdot 2^{\frac{s}{3}} + A_L \cdot B_L) \cdot (2^{\frac{2s}{3}} + 2^{\frac{s}{3}} + 1) +\\ (A_H + A_M) \cdot (B_H + B_M) \cdot 2^s + (A_H + A_L) \cdot (B_H + B_L) \cdot 2^{\frac{2s}{3}} +\\ (A_M + A_L) \cdot (B_M + B_L) \cdot 2^{\frac{s}{3}}\end{aligned} \tag{17}$$

The pseudo code for 571-bit polynomial multiplication is available in Algorithm 24. 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 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.

Algorithm 24 571-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: $C = C_H \oplus C_M \oplus C_L$
```

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 25.

Algorithm 25 571-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:     $\{C[i+1]||C[i]\} = A[i] \times_{64} A[i]$
```

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 $r(z)$. 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 26. 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 26 Binary Field Reduction over $\mathbb{F}_{2^{571}}$

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

Binary Field Inversion For fast and secure against timing attack, we used the Itoh-Tsujii algorithm [35], 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 27. The inversion operation requires only 570 multiplication and

13 squaring operations. For implementation, we conduct the multiplication and squaring in assembly and combine the both operations in C language.

Algorithm 27 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}
```

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.

5.3 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 14, 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 [26] 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.

6 Ring-LWE Implementation over AVR

Today's widely used public-key cryptosystems are mainly based on integer factorization and discrete logarithm problems, which are believed to be intractable with classical computers. However, these hard problems can be solved by using Shor's algorithm [75] and its variant on a quantum computer. Lattice-based cryptography is considered as a premier candidate for post-quantum cryptosystems. Its security is based on worst-case computational assumptions in lattices that remain hard even for quantum computers. The Internet is currently in the midst of a transition from a network that connects commodity computers (e.g. PCs, laptops) to a network of smart objects ("things"). Even today, there are more "non-traditional" computing devices connected to the Internet than "conventional" computers [21]. Among the smart devices that are (or will soon be) populating the Internet are all kinds of sensors, actuators, meters, consumer electronics, medical monitors, household appliances, vehicles, and even items of clothing. Many of these devices are very constrained in terms of computing power and memory resources. For example, a typical wireless sensor node, such as the widely-used MICAz mote, features an 8-bit AVR ATmega processor clocked at 8 MHz and a few KB RAM. However, in order to communicate

Table 14: Comparison results of binary field arithmetic, scalar multiplication and ECDH over K-571

Algorithm	Architecture	Processor	Clock Cycles
Multiplication			
LD [13]	Cortex-A8	ARMv7	3,071
LD [13]	Cortex-A9	ARMv7	3,140
LD [13]	Cortex-A15	ARMv7	1,424
KNV [13]	Cortex-A8	ARMv7	1,506
KNV [13]	Cortex-A9	ARMv7	1,889
KNV [13]	Cortex-A15	ARMv7	1,103
Proposed Method (KNP)	**Apple-A7**	**ARMv8**	**132**
Squaring			
Table [13]	Cortex-A8	ARMv7	349
Table [13]	Cortex-A9	ARMv7	394
Table [13]	Cortex-A15	ARMv7	282
VMULL [13]	Cortex-A8	ARMv7	126
VMULL [13]	Cortex-A9	ARMv7	146
VMULL [13]	Cortex-A15	ARMv7	99
Proposed Method (PMULL)	**Apple-A7**	**ARMv8**	**41**
Inversion (Itoh-Tsujii)			
Previous Method [13]	Cortex-A8	ARMv7	90,936
Previous Method [13]	Cortex-A9	ARMv7	97,913
Previous Method [13]	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 [13]	Cortex-A8	ARMv7	4,870,000
Previous Method [13]	Cortex-A9	ARMv7	6,018,000
Previous Method [13]	Cortex-A15	ARMv7	3,603,000
Proposed Method	**Apple-A7**	**ARMv8**	**783,705**

securely with such devices, they need to be able to execute public-key cryptography as otherwise end-to-end authentication and end-to-end key establishment would not be possible. Implementing public-key algorithms on 8-bit processors poses a big challenge, not only for RSA and ECC, but also post-quantum techniques like lattice-based cryptography. Therefore, it is necessary to study how well "cryptosystems of the future" are suited for the "Internet of the future." In other words, it is necessary to study how well lattice-based cryptography can be implemented on 8-bit processors such as the AVR series processors [3].

The introduction of learning with errors (LWE) problem [63] and its ring variant (ring-LWE) [50] provide an efficient way to build lattice based public key cryptosystems. The first practical evaluations of LWE and ring-LWE based encryption schemes were presented by Göttert et al. in CHES'12 [24]. In the experimental results, the ring-LWE based encryption scheme is faster by at least a factor of four and requires less memory in comparison to the encryption scheme based on the standard LWE problem. The following hardware or software implementations [57, 16, 10, 9, 62] of ring-LWE based public-key encryption or digital signature schemes improved performance and memory requirements. Oder et al. in [57] presented an efficient implementation of Bimodal Lattice Signature Schemes (BLISS) on a 32-bit ARM Cortex-M4F microcontroller; The most optimal variant of their implementation cost $6M$ cycles for signing, $1M$ cycles for verification and $368M$ cycles for key generation, respectively, at a medium-term security level. Recently, De Clercq et al. in [16] implemented ring-LWE encryption scheme on the identical ARM processors, their implementation required $121K$ cycles per encryption and $43.3K$ cycles per decryption at medium-term security level while $261K$ cycles per encryption and roughly $96.5K$ cycles per decryption for long-term security level. The first time that a lattice-based cryptographic scheme was implemented on an 8-bit processor belonged to Boorghany et al. in [10, 9]. In [10, 9], the authors evaluated four lattice-based authentication protocols on both 8-bit AVR and 32-bit ARM processors. In particular, for 8-bit AVR implementation, their implementation needed $754,668$ cycles and $2,207,787$ cycles for Fast Fourier Transform (FFT) transform at medium-term and long-term security level, respectively. Based on efficient implementation of polynomial multiplication and Gaussian sampler function, their implementation of LWE based encryption scheme required $2,770,592$ clock cycles for key generation, $3,042,675$ clock cycles for encryption as well as $1,368,969$ clock cycles for decryption at medium-term security level. Recently, Pöppelmann et al. [62] studied and compared implementations of Ring-LWE encryption and the Bimodal Lattice Signature Scheme (BLISS) on an 8-bit Atmel ATxmega128 microcontroller. For medium-term security level, they achieved $1,314,977$ cycles and $381,254$ for ring-LWE encryption and decryption operations, respectively. Recently in CHES'15, Zhe et al. presented two implementations of ring-LWE encryption scheme for both medium-term and long-term security levels on an 8-bit AVR processor [47]. The first one is high-speed (HS) oriented, while the second is memory-efficient (ME) oriented. For medium-term security level, the former one only requires $590K$, $666K$ and $299K$ clock cycles for key-generation, encryption and decryption, respectively. Similarly for long-term security level, the key-generation, encryption and decryption of HS implementation take $2.3M$, $2.7M$ and $700K$ clock cycles, respectively. Both of the HS and ME implementations significantly improve the speed records for ring-LWE encryption scheme on 8-bit AVR processors.

Research Contributions This paper continues the line of research on the efficient implementation of the ring-LWE encryption scheme on an 8-bit AVR processor. The core contributions are several optimizations to reduce the execution time and running ROM requirements of ring-LWE encryption scheme. More specifically, our contributions are listed as follows:

1. The efficiency of coefficient modular multiplication is a pre-requisite for high-speed NTT operation. We propose the Montgomery algorithm based reduction operation. The approach performs replaced a number of shift and addition operations by few number of `mul` operation.
2. To minimize the ROM requirements of our discrete Gaussian sampler based on the Knuth-Yao random walk algorithm [37], we propose an optimal LUT representation.

6.1 Background

The Ring-LWE Encryption Scheme The encryption schemes we use in this paper are based on the ring version of the learning with errors (ring-LWE) problem. The more general form of the problem, i.e. the LWE problem is parameterized by a dimension $n \geq 1$, a modulus q, and an error distribution. The error distribution is generally taken as a discrete Gaussian distribution $\mathcal{X}_\sigma$ with standard deviation σ and mean 0 to achieve best entropy/standard deviation ratio [17]. In the literature the LWE problem is defined as following:

Two polynomials $\mathbf{a}$ and $\mathbf{s}$ are chosen uniformly from $\mathbb{Z}_q^n$. The first polynomial is a global polynomial, whereas the second polynomial is kept as a secret. The LWE distribution $A_{s,\mathcal{X}}$ is defined over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ and comprises of the elements $(\mathbf{a}, t)$ where $t = \langle \mathbf{a}, \mathbf{s} \rangle + e \bmod q \in \mathbb{Z}_q$ for some error polynomial e sampled from the error distribution $\mathcal{X}_\sigma$. In the *search* version of the LWE problem, an attacker is provided a polynomial number of $(\mathbf{a}, t)$ pairs sampled from $A_{s,\mathcal{X}}$ and he (she) tries to find the secret polynomial $\mathbf{s}$. Similarly in the *decision* version of the LWE problem, an attacker tries to distinguish between a polynomial number of samples from $A_{s,\mathcal{X}}$ and the same number of samples from $\mathbb{Z}_q^n \times \mathbb{Z}_q$.

In 2010, Lyubashevshy et al. proposed an encryption scheme based on a more practical algebraic variant of the LWE problem defined over polynomial rings $R_q = \mathbb{Z}_q[\mathbf{x}]/\langle f \rangle$ with an irreducible polynomial $f(x)$ and a modulus q. In the ring-LWE problem, the elements a, s and t are polynomials in the ring R_q. The ring-LWE encryption scheme proposed by Lyubashevshy et al. was later optimized in [64]. Roy et al.'s variant aims at reducing the cost of polynomial arithmetic. In particular, the polynomial arithmetic during a decryption operation requires only one Number Theoretic Transform (NTT) operation. Beside this computational optimization, the scheme performs sampling from the discrete Gaussian distribution using a Knuth-Yao sampler. In the next subsubsection we will first present the mathematical concepts of the NTT and the Knuth-Yao sampling operations and then we will describe the steps used in the Roy et al's version of the encryption scheme.

The Encryption Scheme In this subsection we describe the steps used in the encryption scheme proposed by Roy et al. [64]. We denote the NTT of a polynomial a by $\tilde{a}$.

– The key generation stage **Gen**$(\tilde{a})$: Two error polynomials $r_1, r_2 \in R_q$ are sampled from the discrete Gaussian distribution $\mathcal{X}_\sigma$ by applying the Knuth-

Yao sampler twice.

$$\tilde{r_1} = NTT(r_1), \tilde{r_2} = NTT(r_2)$$

and then an operation $\tilde{p} = \tilde{r_1} - \tilde{a} \cdot \tilde{r_2} \in R_q$ is performed. The public key is polynomial pair $(\tilde{a}, \tilde{p})$ and the private key is polynomial $\tilde{r_2}$.

– The encryption stage **Enc($\tilde{a}$, $\tilde{p}$, M)**: The input message $M \in \{0,1\}^n$ is a binary vector of n bits. This message is first encoded into a polynomial in the ring R_q by multiplying the bits of message by $q/2$. Three error polynomials $e_1, e_2, e_3 \in R_q$ are sampled from $\mathcal{X}_\sigma$. The ciphertext is computed as a set of two polynomials $(\tilde{C_1}, \tilde{C_2})$:

$$(\tilde{C_1}, \tilde{C_2}) = (\tilde{a} \cdot \tilde{e_1} + \tilde{e_2}, \tilde{p} \cdot \tilde{e_1} + NTT(e_3 + M'))$$

– The decryption stage **Dec($\tilde{C_1}$, $\tilde{C_2}$, $\tilde{r_2}$)**: One inverse NTT is performed to recover M':

$$M' = INTT(\tilde{r_2} \cdot \tilde{C_1} + \tilde{C_2})$$

and then a decoder is used to recover the original message M from M'.

Algorithm 28 Iterative Number Theoretic Transform

```
Require: A polynomial $a(x) \in \mathbb{Z}_q[x]$ of degree $n-1$ and $n$-th primitive $\omega \in \mathbb{Z}_q$ of
  unity
Ensure: Polynomial $a(x) = NTT(a) \in \mathbb{Z}_q[x]$
 1: $a = BitReverse(a)$
 2: for $i$ from 2 by $i = 2i$ to $n$ do
 3:   $\omega_i = \omega_n^{n/i}$, $\omega = 1$
 4:   for $j$ from 0 by 1 to $i/2 - 1$ do
 5:     for $k$ from 0 by $i$ to $n-1$ do
 6:       $U = a[k+j]$
 7:       $V = \omega \cdot a[k+j+i/2]$
 8:       $a[k+j] = U + V$
 9:       $a[k+j+i/2] = U - V$
10:     $\omega = \omega \cdot \omega_i$
11: return $a$
```

Number Theoretic Transform Our implementation adopts the Number Theoretic Transform (NTT) for performing the polynomial multiplication. An NTT can be seen as a variant of Fast Fourier Transform (FFT) but performs in a finite ring $\mathbb{Z}_q$. Instead of using the complex roots of unity, NTT evaluates a polynomial multiplication $a(x) = \sum_{i=0}^{n-1} a_i x^i \in \mathbb{Z}_q$ in the n-th roots of unity ω_n^i for $i = 0, \ldots, n-1$, where ω_n denotes a primitive n-th root of unity. Algorithm 28 shows the iterative version of NTT algorithm, which is originally from Cormen et al. in [15].

As shown in Algorithm 28, the iterative NTT algorithm consists of three nested loops. The outermost loop (i-loop, line 2-11) starts from $i = 2$ and increases by doubling i, and the loop stops when $i = n$, thus it has only $log_2 n$ iterations. In each iteration, the value of twiddle factor ω_i are computed by executing a power operation $\omega_i = \omega_n^{n/i}$, and the value of ω is initialized by 1. Compared to i-loop, the j-loop (line 4-10) executes more iterations, the number of iteration can be seen as a sum of a geometric progression for 2^i where i starts from 0 and has a maximum value of $log_2(n-1)$, thus, the j-loop has $n-1$ iterations. In each iteration of j-loop, the twiddle factor ω is updated by performing a coefficient modular multiplication in line 10. Apparently, the innermost loop (k-loop, line 5-9) occupies most part of the execution time of NTT algorithm since it is executed roughly $\frac{n}{2} log_2 n$ times. In each iteration of the innermost loop (line 6-9), two coefficients $a[i+j]$ and $a[i+j+i/2]$ are loaded from memory into registers, and then $a[i+j+i/2]$ are multiplied by the twiddle factor ω, after that, the value of $a[k+j]$ and $a[k+j+i/2]$ are updated and stored in the memory.

The Gaussian Sampler The ring-LWE cryptosystem needs samples from a discrete Gaussian distribution to provide the error polynomials during the key generation and encryption operations. There are several methods for sampling from a discrete Gaussian distribution. Among them we selected Knuth-Yao algorithm. The Knuth-Yao algorithm stores probabilities of the sample points and performs a random walk by following a binary tree, namely the discrete distribution generating (DDG) tree [37, 65, 20]. A DDG tree efficiently counts the visited non-zero nodes to find the sample based on probability.

Algorithm 29 Knuth-Yao Sampling

```
Require: Probability matrix $P_{mat}$, random number $r$, modulus $q$
Ensure: Sample value $s$
 1: for $col$ from 0 by 1 to $MAXCOL$ do
 2:     $d \leftarrow 2d + (r\&1)$
 3:     $r \leftarrow r \gg 1$
 4:     for $row$ from $MAXROW$ by $-1$ to 0 do
 5:         $d \leftarrow d - P_{mat}[row][col]$
 6:         if $d = -1$ then
 7:             if $(r\&1) = 1$ then
 8:                 return $q - row$
 9:             else
10:                 return $row$
11: return 0
```

As shown in Algorithm 29, the DDG tree is constructed on-the-fly, eliminating the need for storing the entire tree. A random walk is performed from the root of the DDG tree. Each random walk checks a random bit to explore from one level of the tree to the next level. The distance counter d represents the number of intermediate nodes to the right side of the visited node. Each non-zero node that is visited, decrements the distance counter by one. When the distance counter is finally decremented to below zero, the terminal node is found, and the current row number of the probability matrix represents the sample. As the probability matrix only contains the positive half of the Gaussian distribution, a random bit is used to decide the sign of the sample. As our scheme performs all operations modulo q, the negative number is found by $q - row$. Algorithm 29 requires consecutive accesses to elements from different rows in the same column in P_{mat}. To keep the number of memory accesses low, P_{mat} is suggested to be stored in a column-wise form [16]. Another optimized variant of the encryption scheme is called YASHE scheme. YASHE scheme makes efforts to reduce the size of ciphertext, which further reduces the communication cost in the practical usage of ring-LWE.

Parameter Selection Our implementation adopts the parameter sets (n, q, σ) with $(256, 7681, 11.31/\sqrt{2\pi})$ and $(512, 12289, 12.18/\sqrt{2\pi})$ for security levels of 128-bit and 256-bit, respectively. The discrete Gaussian sampler is limited to 12σ to achieve a high precision statistical difference from the theoretical distribution, which is less than 2^{-90}. These parameter sets were also used in most of the previous hardware implementations, e.g., [24, 64] and software implementations, e.g., [10, 9, 16]. This also helps us to compare our work with previous work.

6.2 Optimization Techniques for NTT Computation

In this subsection, we describe the optimization technique to reduce the execution time of NTT and inverse NTT on 8-bit AVR processors.

Montgomery Reduction In the NTT computation, the majority of the execution time is spent on computing reduction operation since it is performed in the innermost k-loop. Thus, fast reduction operation is a perquisite for high-speed implementation of NTT algorithm.

We propose an optimized Montgomery reduction technique for performing the $\bmod 7681$ and $\bmod 12289$ operations. Since shift and addition based SMAS2 method requires a number of shift operations. This main idea is to replace a number of shift and addition operations into few number of multiplication operations. Firstly we conduct multiplying the inverse of modulus to get operands. After then the results are masked with `0x1fff`. The results are multiplied by modulus and then the intermediate results are added to the previously calculated results. Since the modulus is 13-bit wise, we should shift to left by 3-bit to get complete results. Another property of Montgomery technique is its economic register usage; it occupies only 14 out of the 32 available registers [3] such that no `push`/`pop` instructions are required at the beginning/end of the function. SMAS2 method can also be used for modulus $q = 12289$.

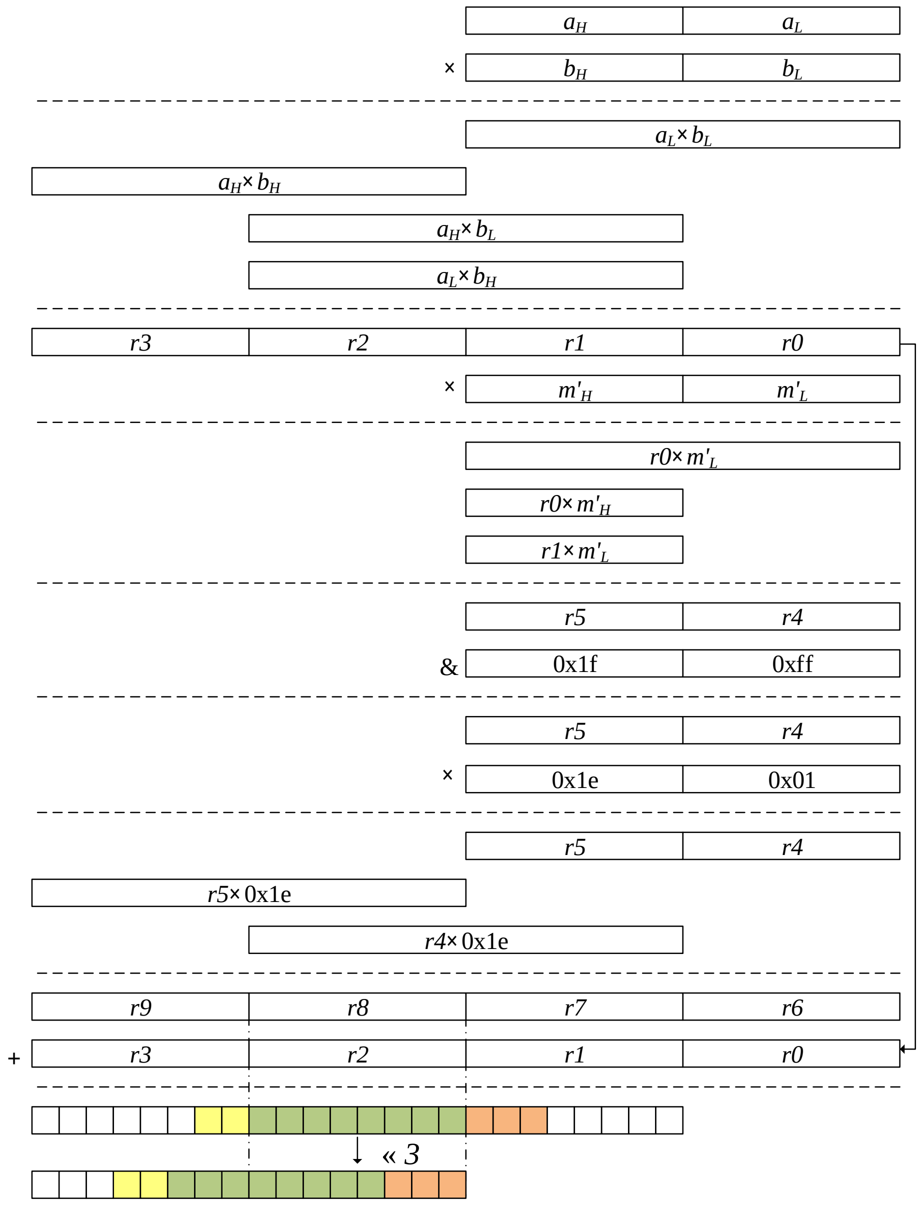

Fig. 4: Montgomery reduction operation for $q = 7681$.

6.3 Optimizations of the Knuth-Yao Sampler

The Knuth-Yao algorithm requires a probability matrix P_{mat}, which contains the probabilities of sampling a random number at a discrete position from the Gaussian distribution. Our Knuth-Yao implementation mainly adopts the optimizations in [16], however, we propose a size optimization by using compressed and index tables.

Probability matrix with small memory consumption. To ensure a precision of 2^{-90} for dimension $n = 256$, the Knuth-Yao algorithm is suggested to have a probability matrix P_{mat} of 55 rows and 109 columns [16]. On an 8-bit AVR processor, we can store each 55-bit column in seven words, where each word is 8-bit long. In this case, only 1-bit is wasted per column and the probability matrix only occupies 6,104 bytes in total. However, the probability matrix contains many leading zero bits. We removed these empty variables and construct the small index look-up table. We firstly scan the column by 8-bit and if there is leading zero bits, we skipped the variables. When we find the setting bit, we marked the starting point of each column into the index table. In order to point out the proper bit location, we needs 3-bit. We saved two index into one 8-bit variables. With this approach we reduce the probability matrix from 672 bytes to 558 bytes.

Efficiently skip the consecutive leading zeros. By constructing the probability table without consecutive leading zeros, we can readily avoid the zero bit execution.

Look-up table in DDG tree. We exploit the look-up table (LUT) approaches proposed in [16] into our byte-wise scanning implementations (shown in lines $1 \sim 9$ of Algorithm 30). First, we perform sampling with an 8-bit random number as an index to the LUT in the first 8 levels for a Gaussian distribution with $\sigma = 11.31/\sqrt{2\pi}$. If the most significant bit of the lookup result is reset, then the algorithm returns the lookup result successfully. Otherwise, the most significant bit of the lookup result is one, then a lookup failure occurs, and the next level of sampling will execute. Similarly, a second LUT will be used for level $9 \sim 13$ in the same Gaussian distribution.

In Algorithm 30, detailed process is described. From Step 3 to 11, LUT based KY sampler is conducted. From Step 12, original KY sampler is conducted. We firstly check the index of row. If count variable is even, we extract the value from lower part of row_{count}. For the other case, we extract the value from higher part of row_{count}. After then, we summate the probability by 8-bit wise and then subtracted from distance. If the distance sets to negative variable, we trace back the variable from last bit to first bit. If we find the case $(d = -1)$, we return the results with current row offset.

Algorithm 30 Knuth-Yao Sampling with byte-wise scanning

```
Require: Probability matrix $P_{mat}$, random number $r$, modulus $q$
Ensure: Sample value $s$
 1: $count \leftarrow 0$
 2: $count_{trace} \leftarrow 0$
 3: $index \leftarrow r\&255$
 4: $r \leftarrow r \gg 8$
 5: $s \leftarrow LUT1[index]$
 6: if $msb(s) = 0$ then
 7:     if $(r\&1) = 1$ then
 8:         return $q - s$
 9:     else
10:         return $s$
11: $d \leftarrow s\&7$
12: for $col$ from 8 by 1 to $MAXCOL$ do
13:     $d \leftarrow 2d + (r\&1)$
14:     $r \leftarrow r \gg 1$
15:     $row_{count} \leftarrow P_{idx}[\text{count++}/2]$
16:     if count is even then
17:         $row_{count} \leftarrow row_{count}\&\text{0x0f}$
18:     else
19:         $row_{count} \leftarrow row_{count} \gg 4$
20:     for $row$ from $row_{count}$ by $-1$ to 1 do
21:         $sum = \sum_{i=count_{trace}}^{count_{trace}+7}(P_{mat}[i])$
22:         $d \leftarrow d - sum$
23:         $count_{trace} \leftarrow count_{trace} + 8$
24:         if $d < 0$ then
25:             if $d = -1$ then
26:                 for $j$ from $count_{trace} - 8$ by 1 to $count_{trace} - 1$ do
27:                     if $P_{mat}[j] = 1$ then
28:                         $j \leftarrow row \times 8 + j$
29:                         if $(r\&1) = 1$ then
30:                             return $q - j$
31:                         else
32:                             return $j$
33:             else
34:                 for $j$ from $count_{trace} - 8$ by 1 to $count_{trace} - 1$ do
35:                     $d \leftarrow d + P_{mat}[j]$
36:                     if $d = -1$ then
37:                         $j \leftarrow row \times 8 + j$
38:                         if $(r\&1) = 1$ then
39:                             return $q - j$
40:                         else
41:                             return $j$
42: return 0
```

6.4 Performance Evaluation and Comparison

Experimental Platform Our implementation used ATxmega128A1 processor on an Xplain board as the target platform. This processor has a maximum frequency of 32 MHz, 128 KB flash program memory and 8 KB SRAM. It is a powerful and popular processor with an AES crypto-accelerator and can be used in a wide range of applications, such as industrial, hand-held battery applications as well as some medical devices. The implementation is written using a mixed ANSI C and Assembly languages. In particular, the main structure of ring-LWE scheme and interface are written in C while the modular operations are implemented in Assembly. We complied our implementation with speed optimization option -O3 on Atmel Studio 6.2. In order to obtain accurate timings, we ran each operation at least 1000 times and calculated the average cycle count for one operation.

Experimental Results Table 15 summarizes the execution time of the main components of ring-LWE encryption schemes (including the NTT, the Knuth-Yao sampler, key-generation, encryption as well as decryption operation) for both of medium-term and long-term security levels.

Table 15: Execution time of main components of the ring-LWE encryption scheme (in clock cycles)

Implementation	NTT	KY	`Gen`	`Enc`	`Dec`
256	209,318	26,763	637,363	725,667	315,024
512	420,544	255,218	2,062,132	2,492,818	640,609

As shown in the Table 15, the NTT operation only requires $209,318$ clock cycles for 256 implementation, however, the execution time increases sharply to $420,544$ cycles for 512 implementation, which is 2 times slower. The Knuth-Yao sampler for 256 requires an average of $26,763$ cycles, while $255,218$ cycles are needed for 512. Both of these observations can be explained by the fact that the length of coefficients for 512 is twice as 256 and AES accelerator can boost performance of random number generations. Taking 256 as an example, the key generation, encryption and decryption require an execution time of roughly $637K$, $725K$ and $315K$, respectively.

Table 16: Memory requirements of key generation, encryption as well as decryption (in bytes)

Implementation	`Gen`	`Enc`	`Dec`	`Total`
RAM/ROM (256)	1,585/8,770	2,609/8,698	1,585/5,912	2,609/13,490
RAM/ROM (512)	3,121/11,960	6,193/13,372	3,121/8,398	6,193/18,780

Table 16 lists the RAM and ROM requirements of key-generation, encryption and decryption. For the whole ring-LWE encryption scheme implementation, the 256 requires roughly $2.6K$ RAM and $13.5K$ ROM. Both of the RAM requirements increase approximately 130% when comparing 512 with 256.

Table 17: Performance comparison of software implementation of lattice-based cryptosystems on different processors.

Implementations	`NTT/FFT`	`Sampling`	`Gen`	`Enc`	`Dec`
Implementations on desktop processors, e.g., Core 2 Duo:					
Göttert et al. [25] (256)	N/A	N/A	9,300,000	4,560,000	1,710,000
Göttert et al. [25] (512)	N/A	N/A	13,590,000	9,180,000	3,540,000
Implementations on 32-bit ARM processors, e.g., Cortex-M4F:					
DeClercq et al. [16] (256)	31,583	7,296	117,009	121,166	43,324
DeClercq et al. [16] (512)	71,090	14,592	252,002	261,939	96,520
Oder et al. [57] (512)	122,619	935,936	N/A	N/A	N/A
Implementations on 8-bit AVR processors, e.g., ATxmega64, ATxmega128:					
Boorghany et al. [9]	1,216,000	N/A	N/A	5,024,000	2,464,000
Boorghany et al. [10]	754,668	N/A	2,770,592	3,042,675	1,368,969
Pöppelmann et al. [62]	334,646	N/A	N/A	1,314,977	381,254
Zhe et al. [47]	193,731	26,763	589,900	671,628	275,646
This work (256)	209,318	26,763	637,363	725,667	315,024
Boorghany et al. [10]	2,207,787	617,600	N/A	N/A	N/A
Pöppelmann et al. [62]	855,595	N/A	N/A	3,279,142	1,019,350
Zhe et al. [47]	441,572	255,218	2,165,239	2,617,459	686,367
This work (512)	420,544	255,218	2,062,132	2,492,818	640,609

Comparison with Related Work Table 19 compares software implementations of lattice-based cryptosystems on different processors. For the 8-bit AVR platform, the previous work [10, 9, 62] and our implementation adopt the same parameter sets as we mentioned in Subsubsection 6.1. Compared to the recent work [47], our LWE-256 requires more number of clock cycles because Montgomery multiplication over LWE-256 is slower than previous fast reduction algorithm. However, our LWE-512 requires $2.062 \cdot 10^6$, $2.492 \cdot 10^6$ and $0.64 \cdot 10^6$ cycles, which is roughly 1.05X and 1.07X faster. The significant progress achieved is mainly due to the proposed optimizations for speeding up the NTT multiplication and Gaussian sampling computations.

In order to show a comparison of the ring-LWE based encryption scheme with some traditional encryption schemes, we compare our ring-LWE implementation with the state-of-the-art RSA and ECC implementations on 8-bit AVR platform in Table 18. The fastest published RSA implementation belongs to [44], the authors reported an execution time of $76.58 \cdot 10^6$ clock cycles for RSA decryption with 80-bit security level[11]. For a comparison, our LWE-256 only requires 315,024 cycles, which is more than 2556 times faster even with a higher 128-bit security level. They are a few software ECC implementations on 8-bit AVR processors. For example, Düll et al. in [18] reported execution time of 13,900,397 (HS version) and 14,146,844 (ME version) clock cycles for single scalar multiplication using the Curve25519 [5]. The widely used Elliptic Curve Integrated Encryption Scheme (ECIES) is based on scalar multiplication, namely, the encryption requires two scalar multiplications, one with fixed point and the other with random point while the decryption needs one scalar multiplication with random point. For a comparison, our ring-LWE encryption scheme is at least one order of magnitude faster than the ECC work in [18, 48, 2]. These research results also show that the ring-LWE encryption is advantageous to traditional PKC for resource-constraint microncontrollers in case of performance.

Table 18: Comparison of Ring-LWE encryption schemes with RSA and ECC on 8-bit AVR processors (`RAM` and `ROM` in bytes, `Enc` and `Dec` in clock cycles)

Implementation	`PKC`	`RAM`	`ROM`	`Enc`	`Dec`
Gura et al. [30]	RSA-1024	N/A	N/A	3,440,000	87,920,000
Liu et al. [44]	RSA-1024	N/A	N/A	N/A	75,680,000
Düll et al. [18] (ME)	ECC-255	510	9,912	28,293,688	14,146,844
Düll et al. [18] (HS)	ECC-255	494	17,710	27,800,794	13,900,397
Liu et al. [48]	ECC-256	556	14,700	30,539,566	21,118,778
Aranha et al. [2]	ECC-233	3,700	38,600	11,796,480	5,898,240
This work	LWE-256	2,609	13,490	725,667	315,024

7 NTT over ARM-NEON

This paper continues the line of research on the efficient implementation of the Number Theoretic Transform (NTT) on an ARM-NEON processor. The core contributions are several optimizations to reduce the execution time in parallel fashion. More specifically, our contributions are listed as follows:

1. The efficiency of coefficient modular multiplication is a pre-requisite for high-speed NTT operation. We propose the parallel coefficient multiplication for Iterative NTT procedures. The method aims at computing multiple multiplication at once.
2. The modular reduction is the most time consuming operation. We apply the 32-bit wise Shifting-Addition-Multiplication-Subtraction-Subtraction (SMAS2)

[11] To the best of our knowledge, no RSA implementation with 128-bit security level exists on 8-bit AVR processors, thus, we use 80-bit security for a comparison.

approach for reduction operation. The approach performs replaced the expensive `MUL` operation by cheaper shifting and `AND` operations.

3. In the NTT computation, the majority of the execution time is spent on computing the modular reduction operation since it is the most frequent operation in the innermost loop. We exploit the incomplete arithmetic [78] for representing the coefficients and perform the reduction operation in a lazy fashion. Our practical results show this approach reduces roughly 6% of the reduction operations in average.

Based on the above optimization techniques, we present high speed implementations of NTT for both medium-term and long-term security levels on an ARM-NEON processors. For medium-term security level, it only requires $27,910$ clock cycles.

7.1 Optimization Techniques for NTT Computation

In this subsection, we describe several optimization techniques to reduce the execution time of NTT on ARM-NEON processors. Throughout the paper, we represent the coefficient of the polynomial using lower-case letters, and each coefficient, for example a, is represented using two bytes a_H for the higher byte, a_L for the lower byte.

In Algorithm 31, vecterized form of NTT operation is described. Unlike normal order of NTT operation described in Algorithm 28, proposed method conducts two different flows one for single multiplication per single operation and the other for multiple multiplication per single operation. The steps from 3 to 12 conduct conventional way of NTT. On the order hands, the rest parts executes the operation in parallel way. Firstly we conduct whole omega values (ω) in aligned array form throughout the Step 14 $\sim$ 16. From Step 19 to 24, inner loops of NTT are conducted in parallel fashion with the array form. The target variables are loaded into registers in arrary form such as U_{array}, V_{array} and ω_{array}. We conduct the four different modular multiplication at once after then pointer p increases by 4 (SIMD width). Lastly The variable arraies are added and subtracted each other to output the results.

Look-Up Table for the Twiddle Factors In each iteration of the i-loop, a new twiddle factor ω (line 3 of Algorithm 28) is computed by performing a modular multiplication. The total number of times a new ω is computed in an NTT operation is n. In each iteration of the j-loop, the twiddle factor ω is computed as shown in line 10 of Algorithm 28. A straightforward computation of $\omega = \omega \cdot \omega_i$ on-the-fly needs to perform both the memory-access operations of ω and ω_i (including of loading and storing) and its coefficient modular multiplication. In total, $n-1$ times of coefficient modular multiplications and $2(n-1)$ times of loading and storing operations are required. The storing all the intermediate twiddle factors ω and ω_i into RAM is cheap over high-end platforms like ARM-NEON.

Algorithm 31 Vecterized Iterative Number Theoretic Transform

```
Require: A polynomial $a(x) \in \mathbb{Z}_q[x]$ of degree $n-1$ and $n$-th primitive $\omega \in \mathbb{Z}_q$ of
    unity
Ensure: Polynomial $a(x) = NTT(a) \in \mathbb{Z}_q[x]$
 1: $a = BitReverse(a)$
 2: for $i$ from 2 by $i = 2i$ to $n$ do
 3:     $\omega_i = \omega_n^{n/i}$, $\omega = 1$
 4:     if $i = 2$ or $i = 4$ then
 5:         for $j$ from 0 by 1 to $i/2 - 1$ do
 6:             for $k$ from 0 by $i$ to $n - 1$ do
 7:                 $U = a[k + j]$
 8:                 $V = \omega \cdot a[k + j + i/2]$
 9:                 $a[k + j] = U + V$
10:                 $a[k + j + i/2] = U - V$
11:             $\omega = \omega \cdot \omega_i$
12:     else
13:         $\omega_{array}[0] = \omega$
14:         for $p$ from 1 by 1 to $i/2 - 1$ do
15:             $\omega = \omega \cdot \omega_i$
16:             $\omega_{array}[p] = \omega$
17:         for $k$ from 0 by $i$ to $n - 1$ do
18:             $p = 0$
19:             for $j$ from 0 by 4 to $i/2 - 1$ do
20:                 $U_{array} = a[k + j : k + j + 3]$
21:                 $V_{array} = \omega_{array}[p : p + 3] \cdot a[k + j + i/2 : k + j + 3 + i/2]$
22:                 $p = p + 4$
23:                 $a[k + j : k + j + 3] = U_{array} + V_{array}$
24:                 $a[k + j + i/2 : k + j + 3 + i/2] = U_{array} - V_{array}$
25: return $a$
```

Both of the computations of the power of ω_n in i-loop and twiddle factor $\omega = \omega \cdot \omega_i$ in j-loop can be considered as fixed costs. We can pre-compute the all twiddle factors ω into RAM. We only need to transfer the twiddle factor that is required for the current iteration of the j-loop from ROM to RAM. In this way, we do not require to compute the power of ω_n in i-loop and the coefficient modular multiplication in j-loop. As we described before, we stored the twiddle factor in aligned array variable in order to avoid the inefficient memory access costs.

Efficient Coefficient Multiplication The coefficient multiplication is one of the most performance-critical operations of NTT computation in terms of "computational complexity" since each NTT computation requires $\frac{n}{2}log_2n$ coefficient multiplications.

In our implementation, the coefficient is at most 13-bit long, which can each be kept in one 32-bit register. Even we can store two coefficients into one register. However, 13-bit wise multiplication outputs at most 26-bit so we decide to store 13-bit coefficient into one 32-bit register. The ARM-NEON register can contain four 32-bit wise variables. We loaded four different aligned variables and then conduct the multiplication with single multiplication operation.

Fast Reduction In the NTT computation, the majority of the execution time is spent on computing reduction operation since it is performed in the innermost k-loop. Thus, fast reduction operation is a perquisite for high-speed implementation of NTT algorithm.

We propose an optimized SMAS2 reduction technique for performing the $\bmod 7681$ operation. This main idea is to first estimate the quotient of $t = \frac{a}{q}$, and then perform the subtraction $a - t \cdot q$. Finally, the correct result can be obtained by a correction process with a final subtraction. Observing that $2^{13} \equiv 2^9 - 1 \bmod 7681$, it is not difficult to conjecture the approximating value of t is $(a \gg 13) + (a \gg 17) + (a \gg 21)$. More precisely, the reduction process consists of four different basic operations, namely, 32-bit wise Shifting $\rightarrow$ Addition $\rightarrow$ Multiplication $\rightarrow$ Subtraction $\rightarrow$ Subtraction (SAMS2). As shown in Figure 5, we keep the product in one registers ($r0$). The $r0$ is 32-bit long. The colorful parts mean that this bit has been occupied while the white part means the current bit is empty. The reduction with 7681 using SAMS2 approach can be performed as follows:

1. Shifting. We right shift $r0$ by 13-bit, 17-bit and 21-bit. This outputs results $t0$, $t1$ and $t2$.
2. Addition. We then perform the addition of $t0 + t1 + t2$. Apparently, the sum result is less than 16-bit, which can be kept in one register.
3. Multiplication. The third step is to multiply the constant `0x1e01` by $(t0 + t1 + t2)$, which is a 16×13-bit multiplication.
4. Subtraction. Thereafter, we subtract both the sum of $t0 + t1 + t2$ and the product obtained from Step 3 from r.

5. Subtraction. However, the result we get in step 4 may still be larger than $p = 7681$, thus, we do the correction by subtracting the modulus p once.

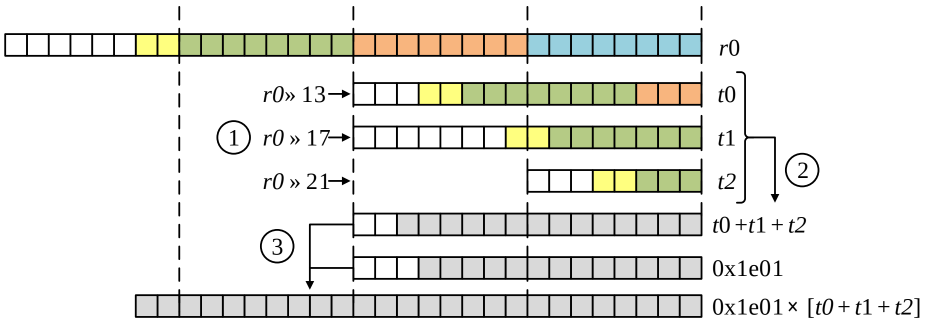

Fig. 5: Fast reduction operation with 32-bit wise SMAS2 method for $q = 7681$. ①: shifting; ②: addition; ③: multiplication; ④: subtraction.

Another property of SMAS2 technique is its economic register usage; it occupies only 14 out of the 32 available registers [3] such that no `push`/`pop` instructions are required at the beginning/end of the function. SMAS2 method can also be used for modulus $q = 12289$.

Reduce the Number of Modular Reduction Operations Besides the coefficient multiplications, addition and subtraction operations are also performed in the innermost loop of NTT. In general, a coefficient addition $r = a + b \bmod p$ (resp. subtraction) can be performed by an addition (resp. subtraction) operation followed by a conditional subtraction (resp. addition) with the prime p. The intermediate result is kept in the range of $[0, p]$.

Inspired by the incomplete modular arithmetic [78], our implementation does not perform an exact comparison between r and p, but rather tolerate an incompletely reduced coefficient $r \in [0, 2^{\lceil log_2 p \rceil}]$. Taking $p = 7681$ as an example, the incomplete coefficient addition works as follows. We first perform a normal coefficient addition, after that, we conduct the 13-bit shift to the right and perform the modular reduction by multiplying the modulus with the shifted results. The intermdeiate results are placed within $[0, 2^{\lceil log_2 p \rceil}]$ [42]. In the very last outermost iteration of NTT (i.e. in our case, $i = 256$), a correction process is performed to bring the final result back into the range $[0, p]$. The incomplete coefficient technique can be used for coefficient addition, subtraction as well as coefficient multiplication. Our practical results show this approach reduces roughly 6% of the modular reduction operations, thus resulting in a speed up of the execution time for the NTT computation.

In Algorithm 32, pseudo codes for vecterized NTT computation is described. Firstly a fair of coefficients and twiddle factors are multiplied and then shift

Algorithm 32 Pseudo codes of vecterized NTT computation

Require: Four 32-bit operands $A[0:3]$(q2), $B[0:3]$(q3), ω(q1), modulo(0x1e01, q0).
Ensure: Eight 32-bit results C(q5,q10).

```
  vmul.i32 q3, q3, q1
  vshr.u32 q4, q3, #13
  vshr.u32 q5, q3, #17
  vshr.u32 q6, q3, #21
  vadd.i32 q4, q4, q5
  vadd.i32 q4, q4, q6
  vmls.i32 q3, q4, d0[0]
  vshr.u32 q4, q3, #13
  vmls.i32 q3, q4, d0[0]
  vadd.i32 q5, q2, q3
  vshr.u32 q4, q5, #13
  vmls.i32 q5, q4, d0[0]
  vshl.i32 q1, q0, #2
  vadd.i32 q2, q2, q1
  vsub.i32 q10, q2, q3
  vshr.u32 q14, q10, #13
  vmls.i32 q10, q14, d0[0]
```

to right by 13, 17 and 21-bit. The shifted variables are added each other. After then modulus and the intermediate results are multiplied and subtracted (MAS) by using `vmls` operation. After then addition of both coefficients is conducted and then conduct one more reduction with shift and MAS operation. For subtraction, the output could be negative value. The negative variable handling is difficult over unsigned type and grouped variables. We firstly added the value with 4×modulus. After then subtraction is performed and one more reduction is conducted similarly as addition operation.

7.2 Performance Evaluation and Comparison

Experimental Platform The ARM Cortex-A9 is full implementations of the ARMv7 architecture including NEON engine. Register sizes are 64-bit and 128-bit for double(D) and quadruple(Q) word registers, respectively. Each register provides short bit size computations such as 8-bit, 16-bit, 32-bit and 64-bit. This feature provides more precise operation and benefits to various word size computations. In particular, the main structure of NTT and interface are written in C while the modular operations are implemented in Assembly. We complied our implementation with speed optimization option -O3. In order to obtain accurate timings, we ran each operation at least 1000 times and calculated the average cycle count for one operation.

Experimental Results Table 19 summarizes the execution time of the main components of Number Theoretic Transform for medium-term security level. As shown in the Table 19, the NTT operation only requires 27,910 clock cycles for 256-bit NTT implementation. The results are gained from several optimization techniques.

Table 19: Performance comparison of software implementation of Number Theoretic Transform on different processors.

Implementations	`NTT/FFT`
32-bit ARM-NEON processors, e.g., Cortex-A9:	
This work (256)	27,910
32-bit ARM processors, e.g., Cortex-M4F:	
DeClercq et al. [16] (256)	31,583
DeClercq et al. [16] (512)	71,090
Oder et al. [57] (512)	122,619
8-bit AVR processors, e.g., ATxmega64, ATxmega128:	
Boorghany et al. [9] (256)	1,216,000
Boorghany et al. [10] (256)	754,668
Pöppelmann et al. [62] (256)	334,646
Zhe et al. [47] (256)	193,731
Boorghany et al. [10] (512)	2,207,787
Pöppelmann et al. [62] (512)	855,595
Zhe et al. [47] (512)	441,572

Comparison with Related Work Table 19 compares software implementations of Number Theoretic Transform on different processors. For the 8-bit AVR and 32-bit platforms, the previous work [10, 9, 62, 16, 57] and our implementation adopt the same parameter sets as we mentioned in Subsubsection 6.1. The most suitable comparison target is 32-bit ARM processor. This architecture shares similar ARM instructions of ARMv7. We can show the impact of NEON engine acceleration. Our works improves performance by 11.4 %.

8 Conclusion

This paper presented the representative public key cryptography including RSA, ECC and Ring-LWE over low-end AVR and high-end ARM-NEON processors. The results show that AVR can compute the various PKC efficiently and ARM-NEON can handle massive body of tasks at once. We believe that the results will be practical resource for cryptography engineers.

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. F. Aranha, R. Dahab, J. C. López, and L. B. Oliveira. Efficient implementation of elliptic curve cryptography in wireless sensors. *Advances in Mathematics of Communications*, 4(2):169–187, May 2010.
3. Atmel Corporation. 8-bit ARV® Instruction Set. User Guide, available for download at `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`, July 2008.
4. Atmel Corporation. 8-bit ARV® Microcontroller with 128K Bytes In-System Programmable Flash: ATmega128, ATmega128L. Datasheet, available for download at `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`, June 2008.
5. D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography — PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer Verlag, 2006.
6. D. J. Bernstein. Batch binary edwards. In *Advances in Cryptology-CRYPTO 2009*, pages 317–336. Springer, 2009.
7. D. J. Bernstein, C. Chuengsatiansup, and T. Lange. Curve41417: Karatsuba revisited. In *Cryptographic Hardware and Embedded Systems–CHES 2014*, pages 316–334. Springer, 2014.
8. D. J. Bernstein and P. Schwabe. Neon crypto. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 320–339. Springer, 2012.
9. A. Boorghany and R. Jalili. Implementation and comparison of lattice-based identification protocols on smart cards and microcontrollers. *IACR Cryptology ePrint Archive*, 2014:78, 2014.
10. A. Boorghany, S. B. Sarmadi, and R. Jalili. On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(3):42, 2015.
11. J. W. Bos and M. E. Kaihara. Montgomery multiplication on the cell. In *Parallel Processing and Applied Mathematics*, pages 477–485. Springer, 2010.
12. J. W. Bos, P. L. Montgomery, D. Shumow, and G. M. Zaverucha. Montgomery multiplication using vector instructions. In T. Lange, K. Lauter, and P. Lisonek, editors, *Selected Areas in Cryptography — SAC 2013*, volume 8282 of *Lecture Notes in Computer Science*, pages 471–489. Springer Verlag, 2014.
13. 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.
14. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, Dec. 1990.
15. T. H. Cormen. *Introduction to algorithms.* MIT press, 2009.
16. R. De Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Efficient software implementation of ring-lwe encryption. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 339–344. EDA Consortium, 2015.
17. L. Ducas. *Lattice based signatures: Attacks, analysis and optimization.* PhD thesis, 2013. Available for download at `http://cseweb.ucsd.edu~lducas/Thesis/index.html`.

18. M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe. High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography*, pages 1–22, 2015.
19. M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe. High-speed curve25519 on 8-bit, 16-bit and 32-bit microcontrollers. *Design, Codes and Cryptography*, to appear. Document ID: bd41e6b96370dea91c5858f1b809b581, `http://cryptojedi.org/papers/\#mu25519`.
20. N. C. Dwarakanath and S. D. Galbraith. Sampling from discrete gaussians for lattice-based cryptography on a constrained device. *Applicable Algebra in Engineering, Communication and Computing*, 25(3):159–180, 2014.
21. D. Evans. The Internet of things: How the next evolution of the Internet is changing everything. Cisco IBSG white paper, available for download at `http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`, apr 2011.
22. S. for Efficient Cryptography Group. Recommended elliptic curve domain parameters. 2000.
23. Free Software Foundation, Inc. GMP: The GNU Multiple Precision Arithmetic Library. Available for download at `http://www.gmplib.org/`, Feb. 2015.
24. N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 512–529. Springer, 2012.
25. N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 512–529. Springer, 2012.
26. C. P. Gouvêa and J. López. Implementing gcm on armv8. In *Topics in Cryptology—CT-RSA 2015*, pages 167–180. Springer, 2015.
27. R. Granger and M. Scott. Faster ecc over\ mathbb {F} _ {2^{521}-1}. In *Public-Key Cryptography–PKC 2015*, pages 539–553. Springer, 2015.
28. 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.
29. S. Gueron and V. Krasnov. Software implementation of modular exponentiation, using advanced vector instructions architectures. In *Arithmetic of Finite Fields*, pages 119–135. Springer, 2012.
30. N. Gura, A. Patel, A. S. Wander, H. Eberle, and S. Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer Verlag, 2004.
31. M. Hamburg. Ed448-goldilocks, a new elliptic curve.
32. M. Hutter and P. Schwabe. Multiprecision multiplication on avr revisited. 2014.
33. M. Hutter and E. Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems — CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 459–474. Springer Verlag, 2011.
34. Intel Corporation. Using streaming SIMD extensions (SSE2) to perform big multiplications. Application note AP-941, order number 248606-001, July 2000.
35. 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.

36. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.
37. D. E. Knuth and A. C. Yao. The complexity of nonuniform random number generation. *Algorithms and Complexity: New Directions and Recent Results*, pages 357–428, 1976.
38. Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing montgomery multiplication algorithms. *Micro, IEEE*, 16(3):26–33, 1996.
39. Y. Lee, I.-H. Kim, and Y. Park. Improved multi-precision squaring for low-end risc microcontrollers. *Journal of Systems and Software*, 86(1):60–71, 2013.
40. Y. Lee, I.-H. Kim, and Y. Park. Improved multi-precision squaring for low-end RISC microcontrollers. *Journal of Systems and Software*, 86(1):60–71, 2013.
41. Z. Liu and J. Großschädl. New speed records for montgomery modular multiplication on 8-bit avr microcontrollers. In *Progress in Cryptology–AFRICACRYPT 2014*, pages 215–234. Springer, 2014.
42. Z. Liu and J. Großschädl. New speed records for montgomery modular multiplication on 8-bit AVR microcontrollers. In D. Pointcheval and D. Vergnaud, editors, *The 7th International Conference on Cryptology in Africa — AFRICACRYPT 2014*, volume 8469 of *Lecture Notes in Computer Science*, pages 215–234. Springer Verlag, 2014.
43. Z. Liu, J. Großschädl, and I. Kizhvatov. Efficient and side-channel resistant rsa implementation for 8-bit avr microcontrollers. In *Workshop on the Security of the Internet of Things-SOCIOT*, 2010.
44. Z. Liu, J. Großschädl, and I. Kizhvatov. Efficient and side-channel resistant RSA implementation for 8-bit AVR microcontrollers. In *Proceedings of the 1st International Workshop on the Security of the Internet of Things (SECIOT 2010)*. IEEE Computer Society Press, 2010.
45. Z. Liu, H. Seo, J. Groszschädl, and H. Kim. Reverse product-scanning multiplication and squaring on 8-bit avr processors. In *16th International Conference on Information and Communications Security (ICICS 2014)*. Springer Verlag, 2014.
46. Z. Liu, H. Seo, H. Kim, and J. Großschädl. Reverse product-scanning multiplication on 8-bit avr processors: Tradeoffs between performance, code size and scalability. In *ICICS 2014*. Springer, 2014.
47. Z. Liu, H. Seo, S. S. Roy, J. Großschädl, H. Kim, and I. Verbauwhede. Efficient ring-lwe encryption on 8-bit avr processors.
48. Z. Liu, E. Wenger, and J. Großschädl. MoTE-ECC: Energy-scalable elliptic curve cryptography for wireless sensor networks. In I. Boureanu, P. Owezarski, and S. Vaudenay, editors, *The 12th International Conference on Applied Cryptography and Network Security — ACNS 2014*, volume 8479 of *Lecture Notes in Computer Science*, pages 361–379. Springer Verlag, 2014.
49. J. López and R. Dahab. High-speed software multiplication in f2m. In *Progress in CryptologyINDOCRYPT 2000*, pages 203–212. Springer, 2000.
50. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)*, 60(6):43, 2013.
51. S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.
52. P. Martins and L. Sousa. On the evaluation of multi-core systems with simd engines for public-key cryptography. In *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*, pages 48–53. IEEE, 2014.

53. P. Martins and L. Sousa. Stretching the limits of programmable embedded devices for public-key cryptography. In *Proceedings of the Second Workshop on Cryptography and Security in Computing Systems*, page 19. ACM, 2015.
54. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press Series on Discrete Mathematics and Its Applications. CRC Press, 1996.
55. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.
56. P. L. Montgomery. Five, six, and seven-term karatsuba-like formulae. *Computers, IEEE Transactions on*, 54(3):362–369, 2005.
57. T. Oder, T. Pöppelmann, and T. Güneysu. Beyond ecdsa and rsa: Lattice-based digital signatures on constrained devices. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. ACM, 2014.
58. U. D. of Commerce/N.I.S.T. Federal information processing standards publication 186-2 fips 186-2 digital signature standard.
59. 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.
60. OpenSSL. The open source toolkit for SSL/TLS. Available for download at `https://www.openssl.org/`, Feb. 2015.
61. K. C. Pabbuleti, D. H. Mane, A. Desai, C. Albert, and P. Schaumont. Simd acceleration of modular arithmetic on contemporary embedded platforms. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.
62. T. Pöppelmann, T. Oder, and T. Güneysu. Speed records for ideal lattice-based cryptography on avr.
63. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009.
64. S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact ring-lwe cryptoprocessor. In *Cryptographic Hardware and Embedded Systems–CHES 2014*, pages 371–391. Springer, 2014.
65. S. S. Roy, F. Vercauteren, and I. Verbauwhede. High precision discrete gaussian sampling on fpgas. In *Selected Areas in Cryptography–SAC 2013*, pages 383–401. Springer, 2014.
66. H. Seo and H. Kim. Multi-precision multiplication for public-key cryptography on embedded microprocessors. In *13th International Workshop on Information Security Applications — WISA 2012*, Lecture Notes in Computer Science, pages 55–67. Springer Verlag, 2012.
67. H. Seo and H. Kim. Optimized multi-precision multiplication for public-key cryptography on embedded microprocessors. *International Journal of Computer and Communication Engineering*, 2(3):255–259, 2013.
68. 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.
69. H. Seo, Z. Liu, J. Choi, and H. Kim. Optimized karatsuba squaring on 8-bit avr processors.
70. H. Seo, Z. Liu, J. Choi, and H. Kim. Multi-precision squaring for public-key cryptography on embedded microprocessors. In *Progress in Cryptology–INDOCRYPT 2013*, pages 227–243. Springer, 2013.

71. 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.
72. H. Seo, Z. Liu, J. Großschädl, J. Choi, and H. Kim. Montgomery modular multiplication on arm-neon revisited.
73. H. Seo, Z. Liu, and H. Kim. Efficient arithmetic on arm-neon and its application for high-speed rsa implementation.
74. 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.
75. P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134. IEEE, 1994.
76. 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.
77. C. D. Walter and S. Thompson. Distinguishing exponent digits by observing modular subtractions. In *Topics in CryptologyCT-RSA 2001*, pages 192–207. Springer, 2001.
78. T. Yanık, E. Savaş, and Ç. K. Koç. Incomplete reduction in modular arithmetic. *IEE Proceedings – Computers and Digital Techniques*, 149(2):46–52, Mar. 2002.

Appendix A. Polynomial Addition in Assembly Code

Algorithm 33 571-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 34 571-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]
```

Appendix C. Two Levels of Hybrid Montgomery Reduction

For the long integers, $\frac{n}{2}$-limb of sub-Montgomery reduction can be separated into $\frac{n}{4}$-limb of two sub-multiplication and two Montgomery reduction and we can take advantages of Karatsuba multiplication further. The detailed descriptions of two-level hybrid Montgomery reduction are available in Figure 6. Firstly $\frac{n}{4}$-limb of sub-Montgomery reduction with $Q[0 \sim 3]$ and $M[0 \sim 3]$ is conducted in part ①. After then the part ② conducts $\frac{n}{4}$-limb of conventional multiplication with $Q[0 \sim 3]$ and $M[4 \sim 7]$. Following parts ③ and ④ conduct $\frac{n}{4}$-limb of Montgomery reduction and multiplication, respectively[12]. The following part ⑤ executes the $\frac{n}{2}$-limb of multiplication with $Q[0 \sim 7]$ and $M[8 \sim 15]$. This procedure is iterated from part ⑥ to ⑩. By exploiting the two more $\frac{n}{4}$-limb of Karatsuba multiplication, we can further reduce the multiplication complexity to $\theta(\frac{13n^2}{16} + n)$.

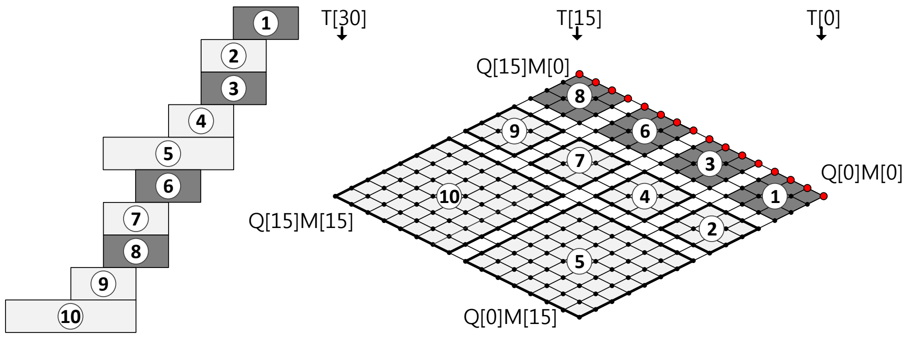

Fig. 6: Two-Level Hybrid Montgomery Reduction

[12] We omit the descriptions of accumulation parts for brief explanations.