Efficient Arithmetic on ARM-NEON and Its Application for High-Speed RSA Implementation

Hwajeong Seo[1], Zhe Liu[2], Johann Großschädl[2], and Howon Kim[1⋆]

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

Abstract. Advanced modern processors support Single Instruction Multiple Data (SIMD) instructions (e.g. Intel-AVX, ARM-NEON) and a massive body of research on vector-parallel implementations of modular arithmetic, which are crucial components for modern public-key cryptography ranging from RSA, ElGamal, DSA and ECC, have been conducted. In this paper, 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.

Keywords: Public-key cryptography, Modular arithmetic, SIMD-level parallelism, Vector instructions, ARM-NEON, RSA

1 Introduction

Multi-precision modular multiplication and squaring are performance-critical building blocks of public-key algorithms (e.g. RSA, ElGamal, DSA and ECC).

⋆ Corresponding Author

One of the most famous modular reduction techniques is Montgomery's algorithm which avoids division in modular multiplication and squaring [22]. 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 [13]. 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 [4], vector instructions on the CELL microprocessor are used to perform multiplication on operands represented with a radix of 2^{16}. In [9], 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 [24]. 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 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 [5]. 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 [19, 20], 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 [2, Section 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 [29]. 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 [5, 19, 20, 29]. 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 [5, 19, 20, 29]. 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 [29, 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 Section 4.1 and performance comparison with related works can be found in Table 1 and 2.
2. *Fast Constant-time Karatsuba multiplication/squaring for ARM-NEON processors.* Inspired by subtractive Karatsuba multiplication [11] and constant-time Karatsuba algorithms on AVR [17], 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 Section 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 Section 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 1 and 2.

The remainder of this paper is organized as follows. In Section 2, we recap the previous best results for squaring on SISD and SIMD architectures. In Section 3, we explore the asymptotically faster integer multiplication algorithm, namely Karatsuba's method. In Section 4, we present novel methods for multi-precision multiplication/squaring and Montgomery algorithms for ARM-NEON engine. Thereafter, we will summarize our experimental results in Section 5. Finally, in Section 6, we conclude the paper.

2 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 $\frac{n^2-n}{2} + n$ whereby $n = \lceil m/w \rceil$, and w and m is the word size and operand length. In the following sub-sections, 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 [15] 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 [28]. SBD method computes doubling using "1-bit left shifting" operation at the end of duplicated partial product computation. Recently, Karatsuba squaring was introduced [27]. 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 $(A_{0\sim255} \times A_{256\sim511})$ with doubling and then conduct other remaining parts ($A_{0\sim255} \times A_{0\sim255}$, $A_{256\sim511} \times A_{256\sim511}$). 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 [3, 2]. 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 [3, 2], 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 [2, Section 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.

3 Karatsuba's Multiplication

One of the multiplication techniques with sub-quadratic complexity is called Karatsuba's multiplication [14]. 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 techniques (e.g. operand-scanning method, product-scanning method, hybrid-scanning method, operand-caching method [21, 6, 10, 12, 25, 26]). 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:

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

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 [1], 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 [11]. The Karatsuba multiplication is readily compatible with

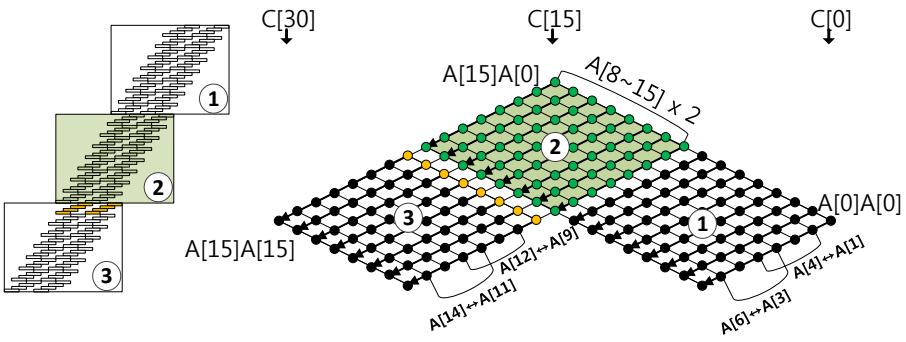

Fig. 1: Double Operand Scanning squaring for SIMD architecture

non-interleaved Montgomery multiplication. In [8], 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. Obviously 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 [5, 19, 20, 29]. 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.

4 Proposed Methods

4.1 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 [15, 28, 27]. 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 1, 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 [29][3]. Taking the 32-bit word with 512-bit squaring as an example, our method works as follows[4]. 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 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`][5], which exceed range of 32-bit in certain

[3] 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.

[4] 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.

[5] 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

Algorithm 1 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

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`)[6]. 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`).

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

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`].

[6] 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`].

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 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 1. 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.

4.2 Constant-Time Karatsuba Multiplication/Squaring

Additive Karatsuba Multiplication The additive Karatsuba's multiplication needs to perform several additions and subtractions (see Section 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 [18]. The smart counter measure is "carry-

Algorithm 2 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: $\{A_{CARRY}, A_{SUM}\} = A_{LOW} + A_{HIGH}$	(SISD)
4: $\{B_{CARRY}, B_{SUM}\} = B_{LOW} + B_{HIGH}$	(SISD)
5: $M = A_{SUM} \cdot B_{SUM}$	**(SIMD)**
6: $M = M + (AND(COM(A_{CARRY}), B_{SUM})) \cdot 2^{\frac{m}{2}}$	(SISD)
7: $M = M + (AND(COM(B_{CARRY}), A_{SUM})) \cdot 2^{\frac{m}{2}}$	(SISD)
8: $M = M + (AND(A_{CARRY}, B_{CARRY})) \cdot 2^{\frac{m}{2}}$	(SISD)
9: $C = L + (M - L - H) \cdot 2^{\frac{m}{2}} + H \cdot 2^m$	(SISD)
10: **return** C	

propagated" addition proposed by [17]. 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 2. 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 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 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) [17]. As like additive Karatsuba method, we

Algorithm 3 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: $\{A_{BORROW}, A_{DIFF}\} = A_{LOW} - A_{HIGH}$ (SISD)
4: $A_{DIFF} = XOR(A_{BORROW}, A_{DIFF})$ (SISD)
5: $A_{DIFF} = A_{DIFF} + COM(A_{BORROW})$ (SISD)
6: $M = A_{DIFF} \cdot A_{DIFF}$ **(SIMD)**
7: $C = L + (L + H - M) \cdot 2^{\frac{m}{2}} + H \cdot 2^{m}$ (SISD)
8: **return** C

Algorithm 4 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

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 3. 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 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.

4.3 Separated Karatsuba Cascade/Double Operand Scanning for Montgomery Multiplication and Squaring

In [29, 5], 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 4, 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 [30]. In order to resolve this problem, in [16], 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 then, the intermediate results (Z) is subtracted by modulus (M)[7]. 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 [31].

5 Results

5.1 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 word size computations. The Cortex-A9 processor is adopted in several devices including iPad 2, iPhone

[7] 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 1: 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	[29]	NEON[5]	ARM[5]	GMP[7]	OpenSSL[23]
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

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.

5.2 Evaluation

We prototyped our methods for ARM Cortex-A9 and A15 processors, which are equivalent to the target processors used in previous works [29, 5, 19, 20]. We compared our results with best previous results from proceeding version of Seo et al.'s paper presented at ICISC 2014 [29]. In Table 1 and 2, we categorize the timings with respect to the architecture that served as experimental platform[8]. 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

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

Table 2: 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	[29]	[19, 20]	NEON[5]	ARM[5]	GMP[7]	OpenSSL[23]
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

provide a squaring specialized method with non-redundant representations on ARM-NEON [29, 5, 19, 20]. We compared our squaring to the latest GNU multiple precision arithmetic library (GMP) ver 6.0.0a [7]. 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 [29, 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[9]. For this reason, our

[9] 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 3: Comparison of proposed implementations with related works

Implementation	Speed Record	Program Style	Scalability	Karatsuba
Published modular multiplication implementations:				
Martins et al. [19, 20]		looped/parameterised	√	
Bos et al. [5]		looped/parameterised	√	
Seo et al. [29]		unrolled		
This work (mul)	√	looped/parameterised	√	√
Published modular squaring implementations:				
This work (sqr)	√	looped/parameterised	√	√

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 [23], 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 3). 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 [29]. 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.

5.3 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 2 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.

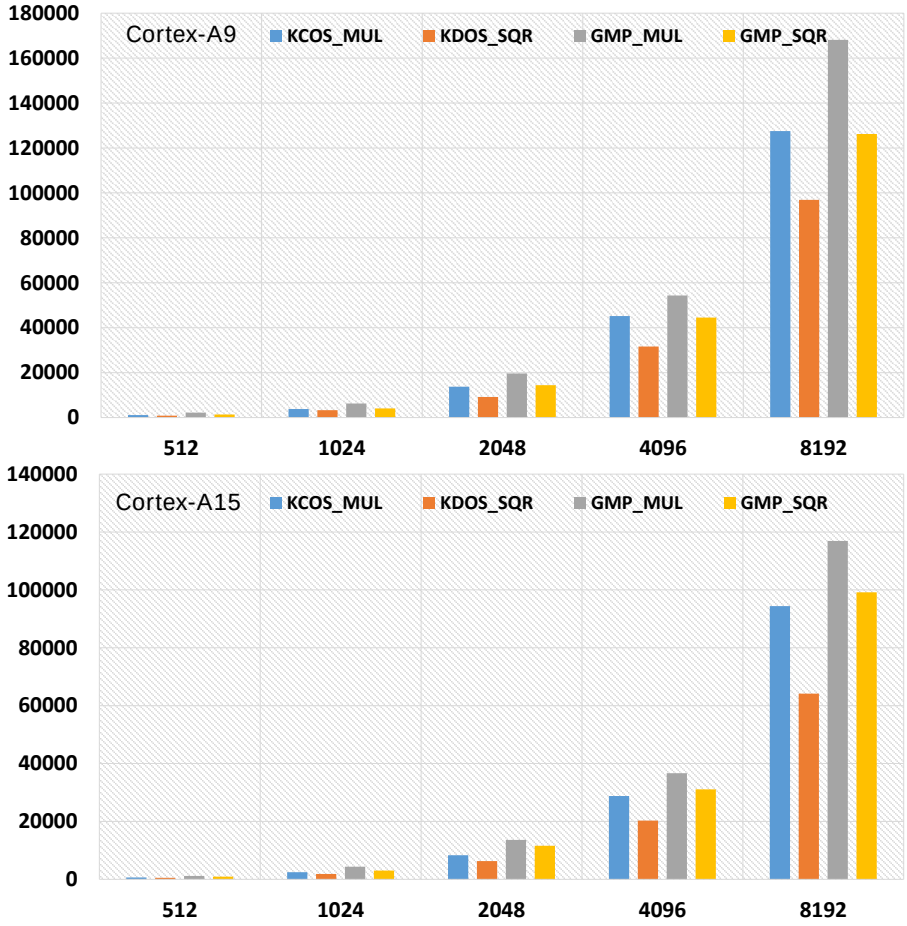

Fig. 2: Results of multiplication/squaring in clock cycles on ARM Cortex-A9/A15

6 Conclusion

We presented optimization techniques to improve the performance of modular arithmetic operations (in particular multiplication, squaring and Montgomery algorithm) on 2-way SIMD platforms. On an ARM Cortex-A15 processor, our separated KCOS and KDOS methods perform 2048-bit Montgomery multiplication and squaring only 19680 and 17584 clock cycles, which are roughly 25% and 41% faster than the NEON implementation of Seo et al. (26232 cycles). For full implementations of 2048-bit RSA encryption and decryption on A15 processor, our implementations only need 367408 and 14250720 clock cycles.

It is also worth to note that our methods are perfectly suitable for processors that support SIMD multiplication (`PMULUDQ`, `VPMULUDQ`) and shuffle (`VPSHUFD`, `PSHUFD`) operations such as Intel-SSE or Intel-AVX family of processors. Based

on these observations, the most obvious future work is to apply the proposed modular multiplication and squaring routines to Intel-SSE and Intel-AVX processors. This will be straight-forward to push the boundaries even further by replacing traditional approaches by our modular arithmetic routines.

References

1. D. J. Bernstein. Batch binary edwards. In *Advances in Cryptology-CRYPTO 2009*, pages 317–336. Springer, 2009.
2. D. J. Bernstein, C. Chuengsatiansup, and T. Lange. Curve41417: Karatsuba revisited. In *Cryptographic Hardware and Embedded Systems–CHES 2014*, pages 316–334. Springer, 2014.
3. D. J. Bernstein and P. Schwabe. Neon crypto. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 320–339. Springer, 2012.
4. J. W. Bos and M. E. Kaihara. Montgomery multiplication on the cell. In *Parallel Processing and Applied Mathematics*, pages 477–485. Springer, 2010.
5. 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.
6. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, Dec. 1990.
7. Free Software Foundation, Inc. GMP: The GNU Multiple Precision Arithmetic Library. Available for download at `http://www.gmplib.org/`, Feb. 2015.
8. 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.
9. 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.
10. 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.
11. M. Hutter and P. Schwabe. Multiprecision multiplication on avr revisited. 2014.
12. 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.
13. Intel Corporation. Using streaming SIMD extensions (SSE2) to perform big multiplications. Application note AP-941, available for download at `http://software.intel.com/sites/default/files/14/4f/24960`, July 2000.
14. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.
15. 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.
16. 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.

17. Z. Liu, H. Seo, J. Großschädl, and H. Kim. Reverse product-scanning multiplication and squaring on 8-bit AVR processors. In L. C.-K. Hui, S. Qing, E. Shi, and S.-M. Yiu, editors, *Information and Communications Security — ICICS 2014*, *Lecture Notes in Computer Science*, Springer Verlag, 2015.
18. S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.
19. 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.
20. 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.
21. 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.
22. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.
23. OpenSSL. The open source toolkit for SSL/TLS. Available for download at `https://www.openssl.org/`, Mar. 2015.
24. 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.
25. H. Seo and H. Kim. Multi-precision multiplication for public-key cryptography on embedded microprocessors. In *Information Security Applications*, pages 55–67. Springer, 2012.
26. 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.
27. H. Seo, Z. Liu, J. Choi, and H. Kim. Optimized karatsuba squaring on 8-bit avr processors.
28. H. Seo, Z. Liu, J. Choi, and H. Kim. Multi-precision squaring for public-key cryptography on embedded microprocessors. In *INDOCRYPT 2013*, pages 227–243. Springer, 2013.
29. H. Seo, Z. Liu, J. Großschädl, J. Choi, and H. Kim. Montgomery modular multiplication on arm-neon revisited.
30. C. D. Walter and S. Thompson. Distinguishing exponent digits by observing modular subtractions. In *Topics in CryptologyCT-RSA 2001*, pages 192–207. Springer, 2001.
31. T. Yanik, E. Savas, and Ç. Koç. Incomplete reduction in modular arithmetic. In *Computers and Digital Techniques, IEE Proceedings-*, volume 149, pages 46–52. IET, 2002.