Multiprecision Multiplication on ARMv8

Zhe Liu*, Kimmo Järvinen†, Weiqiang Liu‡, and Hwajeong Seo§
*APSIA, Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg.
†Department of Computer Science, University of Helsinki, Finland.
‡College of Electronic and Information Engineering, Nanjing University of Aeronautics and Astronautics, China.
§Department of IT, Hansung University, South Korea.
Email: sduliuzhe@gmail.com, kimmo.u.jarvinen@helsinki.fi, liuweiqiang@nuaa.edu.cn, hwajeong@hansung.ac.kr

***Abstract*—Multiplication of large integers is a fundamental operation for public key cryptography. In contemporary public key cryptography, the sizes of integers are typically from more than one hundred bits to even several thousands of bits. Because these sizes exceed the bit widths of all general-purpose processors, these multiplications must be performed with a multiprecision multiplication algorithm which splits the operation into multiple partial products and accumulation steps. To ensure efficiency, multiprecision multiplication algorithms must be designed with special care and optimized for the instruction sets of specific processors. Consequently, developing efficient multiprecision multiplication algorithms and optimizing them for specific platforms has been an active research topic. In this paper, we optimize multiprecision multiplication and squaring specifically for the 64-bit ARMv8 processors which are widely used, for example, in modern smart phones and tablets. We combine the subtractive Karatsuba algorithm, operand-scanning techniques (for multiplication) and sliding-block-doubling methods (for squaring) to accelerate the performance of the 256-bit multiprecision multiplication and squaring by 7.6 % and 7.0 % compared to the OpenSSL implementations. We focus particularly on the multiprecision multiplications that are required in elliptic curve cryptography. Our implementation supports general elliptic curves of various sizes and all source codes are available in public domain.**

***Keywords*-Multiprecision multiplication; public key cryptography; elliptic curve cryptography; 64-bit processor; ARMv8**

I. Introduction

Integer multiplication is the most commonly used operation in public key cryptography (PKC) and, at the same time, amongst the most time-consuming ones. Generally, PKC utilizes integer multiplications with very large integers of several hundreds of bits. They are computed with multiprecision multiplication algorithms that break down the operation into several small partial products which are small enough to be computed with the multiplication instructions of the processor. The partial products are then accumulated in a special way to get the correct product of the large integer multiplication. Because of the frequency and complexity of multiprecision multiplication, its efficiency largely determines the efficiency of the whole PKC implementation. In this paper, we concentrate mostly on multiprecision multiplications that are required for elliptic curve cryptography (ECC) [1], [2] over fields of large prime characteristic, but the techniques may have importance also for other pre-quantum cryptosystems including RSA [3] and cryptosystems based on discrete logarithms (e.g,. El-Gamal [4]) as well as future post-quantum cryptosystems such as Ring Learning with Errors (RLWE) based cryptosystems [5] or Super-Singular Isogeny Diffie-Hellman (SIDH) [6].

The efficiency of multiprecision multiplication is determined mostly by two factors: (1) the size of the partial products, which depends on the bit-width of the processor (e.g., 8 bits for an 8-bit AVR or 32 bits for a 32-bit ARM processor) and also determines the number of partial products that need to be computed and (2) the instructions used for computing the partial products and accumulating them. In particular, certain instruction set architectures (ISA) can include special instructions (e.g., multiply-and-accumulate) that make certain multiprecision multiplication algorithms more efficient. Besides pure efficiency, it is also crucial that multiprecision multiplication algorithm has a constant latency in order to prevent timing side-channel attacks. By combining all the above, we can summarize that optimizing multiprecision multiplication specifically for specific processor architectures is of great importance.

ARM is an ISA for high-performance embedded applications. The most advanced ARM processors, the 64-bit ARMv8 processors, support both 32-bits (AArch32) and 64-bits (AArch64) architectures. The processor includes 31 64-bit registers, which are accessible at any time, and also supports a new instruction set (A64) with 64-bit operands. Compared to its predecessor, ARMv7, it has a more powerful instruction set and more registers to optimize memory access performance. The ARMv8 processors started to dominate the smartphone market soon after the release in 2011 and nowadays they are widely used in various smart phones (e.g., in iPhone and Samsung Galaxy series). Since the processor is used primarily in embedded systems and smart phones, efficient and compact implementations are of special interest. ARMv8 provides two 64-bit multiplication instructions, `MUL` and `UMULH`, both of which carry out one half of a 64×64-bit multiplication (see Fig. 1). In both cases, the inputs are 64-bit registers. `MUL` computes the lower 64-bit half of the results while `UMULH` computes the higher 64-bit half.

Recently, many papers have been published about imple-

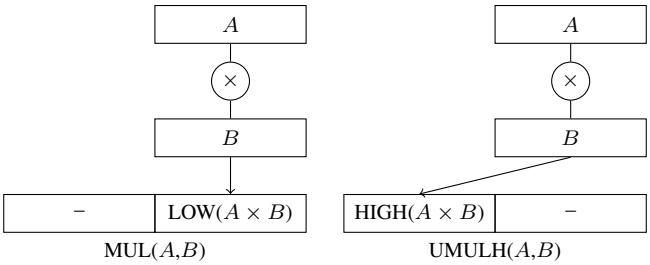

Figure 1: ARMv8 instructions for the 64-bit multiplication: `MUL` and `UMULH`

mentations of cryptography on ARMv8. Gouvêa et al. [7] presented an optimized constant-time implementation of AES-GCM utilizing this instruction set. It achieved very competitive performance: 1.71 cycles per byte for GCM authenticated encryption, 0.51 cycles per byte for GCM authentication and 1.21 cycles per byte for AES-128 encryption. In [8], Seo et al. presented efficient implementations of binary field multiplication in ARMv8. They optimized the multiplication for ARMv8 by combining the Karatsuba algorithm with a 64-bit polynomial multiplication instruction (`PMULL`). Also they achieved very good performance: 57 and 153 cycles for the 251-bit and 571-bit binary fields (B-251 and B-571), respectively. The same authors presented further improvements in [9] by presenting efficient elliptic curve cryptography over B-571 in ARMv8. They improved the binary field multiplication in B-571 from [8] by combining finely aligned multiplication and incomplete reduction techniques with the advantages of the `PMULL` instruction of ARMv8. Despite the fact that ECC over fields of large prime characteristic (e.g., over prime fields) is nowadays significantly more popular than ECC over binary fields, academic papers on efficient implementation of multiprecision integer multiplication for ARMv8 are still missing. The latest OpenSSL library [10] includes the most advanced multiprecision multiplication implementations for the ARMv8 architecture. Even in OpenSSL, only the 256-bit arithmetic required for ECC over the NIST P-256 curve [11] is implemented for the ARMv8 instruction set. The OpenSSL implementation for multiprecision multiplication follows the operand-scanning (schoolbook) approach while multiprecision squaring (multiplication of an operand by itself) follows the sliding-block-doubling method.

In this paper, we study multiprecision multiplication on ARMv8 and introduce several optimizations that lead to significant improvements over the state-of-the-art (i.e., the OpenSSL implementation). We exploit good practices available in the literature and make advantage of the new features in the ARMv8 ISA in order to optimize the multiprecision multiplication for ARMv8. Specifically, we employ the subtractive Karatsuba algorithm and optimize the use of general purpose registers. The detailed implementation techniques are given in Sect. III. Our implementations of multiprecision multiplication provide better performance compared to the OpenSSL implementations in ARMv8. For example, we are 7.6 % and 7.0 % faster for 256-bit multiplications and squarings, respectively. For 512-bit multiplications, we already show improvements of 14.9 % and 10.7 % for multiplication and squaring, respectively.

The remainder of the paper is structured as follows: We review the related work on multiprecision multiplication in Sect. II. We present our contributions and implementations in Sect. III and present the results in Sect. IV. We end with conclusions in Sect. V.

II. Related Work

A. Multiprecision Multiplication

Multiprecision multiplication as well as it efficient implementation have been deeply studied in the past few decades. The n-bit integers A and B are represented in radix-$2w$: $A = \sum_{i=0}^{N-1} A[i]2^{iw}$ and $B = \sum_{j=0}^{N-1} B[j]2^{jw}$ so that both integers decompose into $N = \lfloor n/w \rfloor + 1$ partial operands $A[i]$ and $B[j]$ which are integers from 0 to $2^w - 1$ (w-bit words). Multiprecision multiplication computes the product $A \cdot B$ by computing partial products $A[i] \cdot B[j]$ with different i and j and adds them appropriately to get the final result.

The simplest and most intuitive multiprecision multiplication algorithm is the operand-scanning method (the schoolbook method). As its name suggest, it iterates two nested loops that originate directly from the following equation:

$$C = A \cdot B = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (A[i] \cdot B[j]) 2^{(i+j)w}. \qquad (1)$$

The operand-scanning method performs a multiplication in a row-wise manner so that, first, $A[0]$ is multiplied with all partial operands B, then, the same is done with $A[1]$, etc. Because each partial operand of A is multiplied with all partial operands of B, N^2 partial products are computed in total. Adding all partial products to the corresponding positions produces the final result.

Comba [12] proposed a multiprecision multiplication called product-scanning method where partial products of (1) are computed in a column-wise manner in the order in which they affect the result. I.e., all partial products belonging to the same result word are computed at once: one iterates an index k from 0 to $2N - 2$ and computes partial products $A[i] \cdot B[j]$ for which $i + j = k$ on each iteration. Because of this, all partial products of an iteration are accumulated to the same register and the computation proceeds as follows: $A[0] \cdot B[0]$ is computed first giving the lowest part of the result, then $A[0] \cdot B[1]$ and $A[1] \cdot B[0]$ are computed next and accumulated together with the higher word of the result of $A[0] \cdot B[0]$ to get the next word of the result, etc. This has several advantages. First, since all partial products of each word of the result are computed

and added consecutively, the final result word is obtained directly and no intermediate results have to be stored or loaded in the algorithm. Second, only five working registers are needed to perform the multiplication: two registers for the operands and three registers for the accumulation. This makes the method very suitable for low-resource devices with limited registers.

Gura et al. in [13] proposed a hybrid scanning method that combines the two aforementioned methods. Specifically, the product-scanning and operand-scanning methods are used in the outer and inner loops, respectively. The method uses more registers to store intermediate operands at every iteration of the outer loop which decreases the number of `READ` operations from the memory as a consequence. The performance of the method is determined by a parameter d, which represents the number of `READ` operations in an iteration of the outer loop. Obviously, the method is equal to the product-scanning method if $d = 1$ and to the operand-scanning method if $d = N$.

Scott et al. [14] made an improvement to the original hybrid-scanning method from [13] by employing a set of registers called the carry-catcher. They allow to significantly reduce the number of `MOV` instructions which saves the total number of CPU cycles. In the original hybrid method of [13], carry propagations happen in every iteration of the inner loop when row-wise partial products occur. In the hybrid-scanning method of [14], they happen only in every iteration of the outer loop which results in an increase in performance. The method is particularly useful for squaring (see Sect. II-B) where it results in the fastest schemes available so far.

A variant of the product-scanning method called the operand-caching method was proposed by Hutter et al. in [15]. It follows the principles of the product-scanning method but divides the calculations into several rows. By reordering the sequence of inner and outer row sections, previously loaded operands in working registers are reused for the next partial products. This adds a few `WRITE` instructions, but reduces the number of `READ` instructions and leads to better overall performance. The number of row sections is given by $r = \lfloor n/e \rfloor$, where e denotes the number of words used to cache the words of the operands.

Seo et al. [16] made a further improvement on the operand-caching method from [15]. The improved version named the consecutive operand-caching method uses a caching technique to further reduce the number of memory accesses (`READ` instructions). The key observation was that several memory accesses can be saved because different rows use common operands and it is not necessary to replace all cached values between the rows.

B. Multiprecision Squaring

Squaring is a special case of multiplication where both operands are the same, i.e., $A = B$. All methods discussed above naturally apply for squaring, too. However, there is room for optimizations in multiprecision squaring because certain partial products become the same and need to be performed only once. E.g., $A[0] \cdot B[1] + A[1] \cdot B[0]$ becomes $2A[0] \cdot A[1]$ if $A = B$.

Lee et al. [17] proposed another optimization especially for squaring. In this optimization, the partial products which need to be added twice to the intermediate results, are doubled after they are collected to the accumulator registers at the end of the computation of each column.

Seo et al. [18] gave a further optimization for squaring. By using sliding-block-doubling method, the squaring algorithm executes doubling operation by delaying the operation to the end. The doubling process is conducted on fully accumulated intermediate results with one-bit shift operations. Later, they proposed another method called sliding-middle-block-doubling in [19], which computes the middle parts of the duplicated partial products first and then computes the remaining parts with a doubling process. The technique reduces the number of accesses to the intermediate results.

C. Karatsuba-Ofman Algorithm

We have discussed several improvements for multiprecision multiplication above. All of them optimize the memory access required for computing (1) in some way, but the number of partial products is N^2 for all of them.

In the early 1960s, Karatsuba and Ofman [20] proposed a novel method (called Karatsuba-Ofman or simply Karatsuba algorithm) that reduces the number of partial products at the expense of extra additions. Hence, the Karatsuba algorithm has the potential to perform well in platforms where multiplications are more expensive than additions. The algorithm is based on the remarkable observation that the product $C = A \cdot B$ of two n-bit integers $A = A_L + A_H 2^{n/2}$ and $B = B_L + B_H 2^{n/2}$ can be computed as follows:

$$\begin{aligned} C = A_H \cdot B_H 2^n + ((A_L + A_H) \cdot (B_L + B_H) - \\ A_L \cdot B_L - A_H \cdot B_H) 2^{n/2} + A_L \cdot B_L \end{aligned} \tag{2}$$

As shown above, the Karatsuba algorithm computes a multiplication with only three partial products compared to four that are required by the standard schoolbook multiplication (and the algorithms discussed previously), but requires two additions and two subtractions compared to one addition to compute the middle term (the term corresponding to $2^{n/2}$). For large values of n, the cost of additions and subtractions is insignificant compared to the cost of the multiplications. The procedure may be applied recursively to the intermediate values until some appropriate threshold (e.g., the word size of the processor), after which the classical multiplication (or other method) is employed. The number of partial products can be estimated by $N^{\log_2 3}$, which is a great improvement compared to N^2 of the schoolbook method.

A subtractive variant of the Karatsuba method relies on the fact that the middle term can be expressed by using

Algorithm 1 Subtractive Karatsuba Multiplication

Input: n-bit operands $A = A_L + A_H \cdot 2^{\frac{n}{2}}$, $B = B_L + B_H \cdot 2^{\frac{n}{2}}$
Output: $2n$-bit result $C \leftarrow A \cdot B$
1: $L = L_L + L_H \cdot 2^{\frac{n}{2}} \leftarrow A_L \cdot B_L$ $\{\frac{n}{2}$-bit mul.$\}$
2: $H = H_L + H_H \cdot 2^{\frac{n}{2}} \leftarrow A_H \cdot B_H$ $\{\frac{n}{2}$-bit mul.$\}$
3: $T \leftarrow L_H + H_L$ $\{\frac{n}{2}$-bit add.$\}$
4: $L_H \leftarrow T + L_L$ $\{\frac{n}{2}$-bit add.$\}$
5: $H_L \leftarrow T + H_H$ $\{\frac{n}{2}$-bit add.$\}$
6: $A_D \leftarrow |A_L - A_H|$ $\{\frac{n}{2}$-bit sub.$\}$
7: $B_D \leftarrow |B_L - B_H|$ $\{\frac{n}{2}$-bit sub.$\}$
8: $M = M_L + M_H \cdot 2^{\frac{n}{2}} \leftarrow A_D \cdot B_D$ $\{\frac{n}{2}$-bit mul.$\}$
9: $M \leftarrow L + H - M$ $\{n$-bit add., n-bit sub.$\}$
10: $C \leftarrow L - M \cdot 2^{\frac{n}{2}} + H \cdot 2^n$ $\{n$-bit sub., $\frac{n}{2}$-bit add.$\}$
11: **return** C

Algorithm 2 Subtractive Karatsuba Squaring

Input: n-bit operand $A = A_L + A_H \cdot 2^{\frac{n}{2}}$
Output: $2n$-bit result $C \leftarrow A \cdot B$
1: $L = L_L + L_H \cdot 2^{\frac{n}{2}} \leftarrow A_L \cdot A_L$ $\{\frac{n}{2}$-bit mul.$\}$
2: $H = H_L + H_H \cdot 2^{\frac{n}{2}} \leftarrow A_H \cdot A_H$ $\{\frac{n}{2}$-bit mul.$\}$
3: $T \leftarrow L_H + H_L$ $\{\frac{n}{2}$-bit add.$\}$
4: $L_H \leftarrow T + L_L$ $\{\frac{n}{2}$-bit add.$\}$
5: $H_L \leftarrow T + H_H$ $\{\frac{n}{2}$-bit add.$\}$
6: $A_D \leftarrow |A_L - A_H|$ $\{\frac{n}{2}$-bit sub.$\}$
7: $M = M_L + M_H \cdot 2^{\frac{n}{2}} \leftarrow A_D \cdot A_D$ $\{\frac{n}{2}$-bit mul.$\}$
8: $M \leftarrow L + H - M$ $\{n$-bit add., n-bit sub.$\}$
9: $C \leftarrow L - M \cdot 2^{\frac{n}{2}} + H \cdot 2^n$ $\{n$-bit sub., $\frac{n}{2}$-bit add.$\}$
10: **return** C

absolute values as follows:

$$(A_L + A_H) \cdot (B_L + B_H) - A_L \cdot B_L - A_H \cdot B_H = A_L \cdot B_L + A_H \cdot B_H - |A_H - A_L| \cdot |B_H - B_L| \quad (3)$$

The advantage of the subtractive Karatsuba algorithm is the constant size of operands ($n/2$) for computing partial products, which leads to fast constant-time multiplications. However, the absolute values should implemented with care in two's complement representation. Alg. 1 gives an algorithm for the subtractive Karatsuba multiplication and Alg. 2 shows the corresponding algorithm for squaring.

Recently, Scott [21] denoted that, for the Karatsuba algorithm to be competitive, the actual radix must be a few bits less than the word size in order to facilitate additions without carry processing and, at the same time, to support the ability to distinguish positive and negative numbers. However, this requires an arbitrary degree variant of Karatsuba (ADK) algorithm that allows a non-word size split. The author shows that the total number of multiplications and additions for ADK is less than the numbers required by the operand-scanning method when $N \geq 12$.

III. Our Contributions

In this section, we give our optimized implementations of multiprecision multiplication and squaring on ARMv8 by making the best use of the algorithms discussed above in Sect. II and the specific hardware features of ARMv8.

We begin the description of our implementations with 128-bit multiplication and squaring. Then, we proceed to constructing efficient implementations of 256-bit, 384-bit, and 512-bit multiplication and squaring routines. These bit sizes are relevant, especially, for ECC based PKC which is popular in many applications where ARMv8 is in frequent use. The implementations may have importance also in efficient implementations of, e.g., RSA and post-quantum PKC (e.g., in SIDH) in the future.

A. 128-bit Operations

ARMv8 is a 64-bit processor, but it does not provide a full 64-bit multiplication instruction. The multiplication needs to be carried out with two instructions: `MUL` and `UMULH`. Therefore, some special tricks are needed when implementing multiprecision multiplications with these two instructions.

1) Multiplication: We implemented the 128-bit multiplication by using the subtractive Karatsuba multiplication combined with our implementation tricks. Suppose $A = (A[1], A[0])$ and $B = (B[1], B[0])$ are the 128-bit multiplicand and multiplier, respectively, and they are loaded into four 64-bit registers. First, we compute the lower 64-bit partial product $R_L \leftarrow A[0] \cdot B[0]$. A 64-bit partial product requires one `MUL` and one `UMULH` instruction in order to obtain the full 128-bit result. Second, we compute the higher 64-bit multiplication $R_H \leftarrow A[1] \cdot B[1]$. Third, we perform the subtractions and compute the absolute values to obtain $|A[0] - A[1]|$ and $|B[0] - B[1]|$. In the subtraction, we capture a borrow bit through the `SBC` instruction after the `SUB` instruction. If the borrow bit is captured, the register is set to $2^{32} - 1$. Otherwise, the register is set to 0. The borrow bit indicates whether the sign of the variable is positive (0) or negative ($2^{32} - 1$). Afterwards, we perform a two's complement operation on the subtracted value with the borrow bit by using the `EOR`, `AND` and `ADD` instructions (see Sect. III-E2). The step is performed on both operands and the obtained borrow bits are combined to determine the sign of the last 64-bit multiplication ($R_M \leftarrow |A[0]-A[1]| \cdot |B[0]-B[1]|$) through the two's complement operation. Finally, the result of the 128-bit multiplication is computed via the accumulation step $R_H \cdot 2^{128} + (R_L + R_H - R_M) \cdot 2^{64} + R_L$. In total 13 registers are used in the above process and, hence, the callee-saved registers ($X19 \sim X30$) are not used.

As will be seen in Sect. IV, the above process using the Karatsuba algorithm does not achieve performance enhancements compared to the quadratic variant of the OpenSSL in this case (see Table I). The detailed assembly source code

for the above 128-bit Karatsuba multiplication is available in the Appendix in Alg. 3.

2) Squaring: We use a similar approach also for squaring. Suppose the 128-bit operand is stored into two 64-bit registers. First, we compute the lower 64-bit partial product $R_L \leftarrow A[0] \cdot A[0]$ and, then, the higher 64-bit partial product $R_H \leftarrow A[1] \cdot A[1]$. Second, the subtraction and the absolute value are computed resulting $|A[0] - A[1]|$. The third 64-bit multiplication $R_M \leftarrow |A[0] - A[1]| \cdot |A[0] - A[1]|$ is performed next followed by the the final accumulation step $R_H \cdot 2^{128} + (R_H + R_L - R_M) \cdot 2^{64} + R_L$. The process requires in total 12 registers. Similarly to the 128-bit multiplication, Karatsuba does not give performance enhancements in the case of 128-bit squarings either (see Sect. IV and TableI).

B. 256-bit Operations

1) Multiplication: For the 256-bit multiplication, the operands $A = (A[3], \ldots, A[0])$ and $B = (B[3], \ldots, B[0])$ are stored into eight 64-bit registers. We first compute the lower 128-bit multiplication $R_L \leftarrow A[1 \sim 0] \cdot B[1 \sim 0])$ using the schoolbook method that requires four `MUL`, four `UMULH` and certain additional instructions for accumulating the partial products. Second, we compute the higher 128-bit multiplication $R_H \leftarrow A[3 \sim 2] \cdot B[3 \sim 2]$ similarly. Third, we compute the subtractions and absolute values $|A[1 \sim 0] - A[3 \sim 2]|$ and $|B[1 \sim 0] - B[3 \sim 2]|$ and proceed to the last 128-bit multiplication $R_M \leftarrow |A[1 \sim 0] - A[3 \sim 2]| \cdot |B[1 \sim 0] - B[3 \sim 2]|$. Finally, we obtain the result by performing the accumulation step $R_H \cdot 2^{256} + (R_L + R_H - R_M) \cdot 2^{128} + R_L$. One 256-bit multiplication uses in total 25 registers so that six callee-saved registers ($X19 \sim X24$) are stored into the stack. As will be shown in Sect. IV, the Karatsuba algorithm shows higher performance than quadratic complexity multiplication for the 256-bit multiplication (see Table I).

2) Squaring: The 256-bit operand $A = (A[3], \ldots, A[0])$ is stored into four 64-bit registers. The computation proceeds as above. First, we compute the lower 128-bit multiplication $R_L \leftarrow A[1 \sim 0] \cdot A[1 \sim 0]$ followed by the higher 128-bit multiplication $R_H \leftarrow A[3 \sim 2] \cdot A[3 \sim 2]$. Then, we compute the subtraction and absolute value $|A[1 \sim 0] - A[3 \sim 2]|$ and the 128-bit multiplication $R_M \leftarrow |A[1 \sim 0] - A[3 \sim 2]| \cdot |A[1 \sim 0] - A[3 \sim 2]|$. Finally, the accumulation step $R_H \cdot 2^{256} + (R_L + R_H - R_M) \cdot 2^{128} + R_L$ returns the result. The 256-bit squaring requires in total 19 registers. Similarly as before, the Karatsuba algorithm shows higher performance than quadratic complexity multiplication (see Sect. IV and Table I).

C. 384-bit Operations

1) Multiplication: The 384-bit operands $A = (A[5], \ldots, A[0])$ and $B = (B[5], \ldots, B[0])$ are stored into twelve 64-bit registers. We again begin with the lower and higher 192-bit multiplications $R_L \leftarrow A[2 \sim 0] \cdot B[2 \sim 0]$ and $R_H \leftarrow A[5 \sim 3] \cdot B[5 \sim 3]$, which both require 9 `MUL` and 9 `UMULH` instructions. Also the rest of the multiplication proceeds similarly as before: the subtractions and the absolute values are computed $|A[2 \sim 0] - A[5 \sim 3]|$ and $|B[2 \sim 0] - B[5 \sim 3]|$ followed by the 192-bit multiplication $R_M \leftarrow |A[2 \sim 0] - A[5 \sim 3]| \cdot |B[2 \sim 0] - B[5 \sim 3]|$ and the accumulation step $R_H \cdot 2^{384} + (R_L + R_H - R_M) \cdot 2^{192} + R_L$. The 384-bit multiplication requires in total 31 registers with 12 callee-saved registers ($X19 \sim X30$) which are stored into the stack.

2) Squaring: The 384-bit operand $A = (A[5], \ldots, A[0])$ of the squaring is stored into six 64-bit registers. First, we compute the lower 192-bit multiplication $R_L \leftarrow A[2 \sim 0] \cdot A[5 \sim 3]$ and the higher 192-bit multiplication $R_H \leftarrow A[5 \sim 3] \cdot A[5 \sim 3]$. The computation proceeds with the subtraction $|A[2 \sim 0] - A[5 \sim 3]|$ and the last 192-bit multiplication $R_M \leftarrow |A[2 \sim 0] - A[5 \sim 3]| \cdot |A[2 \sim 0] - A[5 \sim 3]|$. Finally, the accumulation step $R_H \cdot 2^{384} + (R_L + R_H - R_M) \cdot 2^{192} + R_L$ ends the computation. In total 31 registers are used in the 384-bit squaring and 12 callee-saved registers ($X19 \sim X30$) in the stack.

D. 512-bit Operations

1) Multiplication: The operand $A = (A[7], \ldots, A[0])$ and $B = (B[7], \ldots, B[0])$ of the 512-bit multiplication are stored into 16 64-bit registers. Unlike the previous cases, we use 2-level Karatsuba multiplication for the 512-bit multiplication. First, we compute the lower 256-bit multiplication $R_L \leftarrow A[3 \sim 0] \cdot B[3 \sim 0]$ using the 1-level Karatsuba multiplication of two 256-bit operands as described in Sect. III-B1. This 256-bit partial product requires 12 `MUL` and 12 `UMULH` instructions. Second, we compute the higher 256-bit multiplication $R_H \leftarrow A[7 \sim 4] \cdot B[7 \sim 4]$ similarly. Third, we compute the subtractions and absolute values $|A[3 \sim 0] - A[7 \sim 4]|$ and $|B[3 \sim 0] - B[7 \sim 4]|$ and the last 256-bit multiplication $R_M \leftarrow |A[3 \sim 0] - A[7 \sim 4]| \cdot |B[3 \sim 0] - B[7 \sim 4]|$ using the 256-bit 1-level Karatsuba multiplication. Finally, the accumulation step $R_H \cdot 2^{512} + (R_L + R_H - R_M) \cdot 2^{256} + R_L$ gives the result of the full 512-bit multiplication. The above process requires in total 31 registers with 12 callee-saved registers ($X19 \sim X30$) in the stack. Additionally, 16 bytes of the stack are used for intermediate results.

2) Squaring: The 512-bit operand $A = (A[7], \ldots, A[0])$ of the squaring is stored into eight 64-bit registers. For this case, we also use the 2-level Karatsuba approach. First, we compute the lower and higher 256-bit multiplications $R_L \leftarrow A[3 \sim 0] \cdot A[3 \sim 0]$ and $R_H \leftarrow A[7 \sim 4] \cdot A[7 \sim 4]$ with the 1-level 256-bit Karatsuba algorithm described in Sect. III-B1. We obtain the result of the 512-bit squaring by computing the subtraction and absolute value $|A[3 \sim 0] - A[7 \sim 4]|$, the final 256-bit multiplication $R_M \leftarrow |A[3 \sim 0] - A[7 \sim 4]| \cdot |A[3 \sim 0] - A[7 \sim 4]|$, and the accumulation step $R_H \cdot 2^{512} + (R_L + R_H - R_M) \cdot 2^{256} + R_L$. The 512-bit

Table I: A comparison of execution times (clock cycles) and stack sizes (bytes) of multiplication and squaring

		Input size (bits)			
Method		128	256	384	512
Multiplication					
Operand-scanning [10]	cycle	19	66	148	261
	byte	0	32	96	96
This paper	cycle	24	61	126	222
	byte	0	48	96	112
	Karatsuba	1-level	1-level	1-level	2-level
Squaring					
Sliding Block	cycle	17	43	87	149
Doubling [18], [10]	byte	0	0	32	96
This paper	cycle	20	40	84	133
	byte	0	0	96	96
	Karatsuba	1-level	1-level	1-level	2-level

squaring requires in total 31 registers and 12 callee-saved registers ($X19 \sim X30$) which are stored into the stack.

E. Optimizations

1) Generation of the Carry Register: In the accumulation of partial products, the most significant word can generate a carry bit, which should be stored into the higher word. We initialize and use a zero register in order to store the carry bit. The general approach is as follows: `MOV x0, #0`; `ADDS x1, x1, x2`; `ADCS x3, x0, x0`. In the first instruction, we initialize the zero register `x0`. The second instruction computes the addition and the third instruction captures its carry to the register `x3` with the help of the zero register. Alg. 3 in the Appendix shows how the carry register is used in the 128-bit Karatsuba multiplication.

2) Two's Complement: In order to generate a two's complement value, we perform a subtraction and logical operations. The general approach to obtain the two's complement is as follows: `SBCS x2, x2, x2`; `EOR x0, x0, x2`; `AND x2, x2, #1`; `ADD x0, x0, x2`. Alg. 3 in the Appendix shows how the two's complement is computed in the 128-bit Karatsuba multiplication.

IV. Evaluations

We programmed our implementations by using Xcode and benchmarked them on iPad mini 2. The device is equipped with an Apple A7 (APL0698) system-on-chip including a 64-bit ARMv8-A dual-core processor running at the frequency of 1.3 GHz. The programs[1] were written in mixed C and assembly for deep optimizations using the ARMv8-specific features. The code was compiled with `-Ofast` optimization level. The timings are acquired as the number of clock cycles required to execute the codes on a real device and are averaged over 1,000,000 executions.

Table I shows a comparison with the previous works. To the best of our knowledge, there are no academic papers that have presented implementations of multiprecision multiplication and squaring on ARMv8. Therefore, we compared our results with the OpenSSL library [10] which includes implementations of the 256-bit multiplication using the operand-scanning method and the 256-bit squaring using the sliding block doubling method. For comparison purposes, we implemented similar methods also for other bit widths.

For the 128-bit case, the operand-scanning method shows better performance because the Karatsuba algorithm reduces only two multiplication instructions but adds several other instructions (see Table II for the detailed instruction counts). Already for the 256-bit case, the asymptotically faster Karatsuba multiplication and squaring show higher performance than the previous works. For multiplication, we achieved 7.6 %, 14.8 %, and 14.9 % performance enhancements for the 256-, 384-, 512-bit cases, respectively. For squaring, we achieved 7.0 %, 3.4 %, and 10.7 % performance enhancements for the 256-, 384-, 512-bit cases, respectively. The stack sizes are slightly larger than for the previous methods since the Karatsuba algorithm requires a larger number of registers for the intermediate results. However, the additional stack size is reasonable and, typically, does not present a problem in applications of the advanced ARMv8 processors which often include a lot of RAM (e.g., 1–4 GB).

Table II gives a detailed comparisons of instruction counts of multiprecision multiplication and squaring in ARMv8. The main instructions include multiplication, memory access and other arithmetic operations. Particularly, multiplication requires 6 clock cycles and memory access requires 4 clock cycles each. The other arithmetic operations need only 1 clock cycle. If we replace one multiplication with one to five other arithmetic operations, the performance should increase. However, as the results in Table I show, the final latency is not a simple weighted sum of instruction counts due to microarchitectural features such as multiple levels of pipeline as well as parallelism of memory access and arithmetic operations. Consequently, the performance enhancements for particularly the 256- and 384-bit squaring operations are slightly smaller than what could be expected based on Table I. Nonetheless, performance increases are observed for all operations where the operands are at least 256 bits long. We emphasize that multiprecision multiplication and squaring are fundamental operations for PKC and even small improvements for them have significance for the overall performance of cryptographic operations.

V. Conclusion

In this paper, we presented optimized multiprecision multiplication and squaring implementations for the 64-bit ARMv8 processors. Our implementations utilized the subtractive Karatsuba algorithm and ARMv8 specific optimizations. This work shows that our implementations are more efficient than the OpenSSL implementation already for 256-bit operands. Our implementations achieved performance en-

[1]The source codes are available in https://github.com/solowal/ARITH

Table II: A comparison of instruction counts for multiplication and squaring

Operation	ADD/ADC	SUB/SBC	MUL/UMULH	MOV	NEG	EOR/AND	ASR	LDP	STP
Previous operand-scanning multiplication [10]									
128-bit	7	–	8	2	–	–	–	2	2
256-bit	32	1	32	4	–	–	–	6	6
384-bit	77	1	72	6	–	–	–	12	12
512-bit	158	1	128	24	15	–	–	14	14
Proposed Karatsuba multiplication									
128-bit	14	5	6	1	–	8	–	2	2
256-bit	45	8	24	1	–	11	1	7	7
384-bit	84	10	54	1	–	15	1	12	12
512-bit	182	33	72	2	–	52	4	32	21
Previous sliding block doubling squaring[18], [10]									
128-bit	6	–	6	1	–	–	–	1	2
256-bit	27	–	20	4	–	–	–	2	4
384-bit	56	1	42	6	–	–	–	5	8
512-bit	94	1	72	9	1	–	–	10	14
Proposed Karatsuba squaring									
128-bit	7	5	6	1	–	2	–	1	2
256-bit	32	8	18	1	–	3	–	2	4
384-bit	60	12	36	1	–	4	–	9	12
512-bit	124	39	54	2	–	14	–	10	14

ADD/ADC (1 cycle): addition w/o carry / addition w/ carry
SUB/SBC (1 cycle): subtraction w/o borrow / subtraction w/ borrow
MUL/UMULH (6 cycles): multiplication (lower 64-bit)/ multiplication (higher 64-bit)
MOV/NEG (1 cycle): move / negation
EOR/AND/ASR (1 cycle): exclusive-or / logical and / arithmetic right shift
LDP/STP (4 cycles): load pair / store pair

hancements of 7.6 % and 7.0 % for the 256-bit multiplication and squaring, respectively, and even larger improvements for larger operand sizes. These are significant improvements because multiprecision multiplication and squaring are fundamental operations, e.g., in PKC and have a very significant effect on their performance. Our implementations also have constant execution times which is essential in order to avoid side-channel attacks. Since our codes are available in public domain, other cryptography engineers can directly use them for their cryptography applications.

Possible directions for future work are to apply the multiplication and squaring routines described in this paper to ECC. For example, OpenSSL uses quadratic-complexity multiplication and squaring on ARMv8 for operations on the popular Curve25519 elliptic curve [22] and replacing that code with our implementations using the subtractive Karatsuba algorithm will improve the performance of these critical cryptographic operations. Furthermore, this paper focused on ECC friendly operand sizes. It will be of interest to investigate even longer operand sizes that are relevant for RSA implementations. Other possible domains for our implementations include the recent post-quantum PKC implementations (e.g., for the SIDH cryptosystem [6]) which also use multiprecision multiplications with large integers.

ACKNOWLEDGMENTS

The work of K. Järvinen was partly funded by Academy of Finland under project number 303576 and the work of Weiqiang Liu was supported by a grant from the National Natural Science Foundation of China (61401197).

REFERENCES

[1] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology — CRYPTO '85*, ser. Lecture Notes in Computer Science, vol. 218. Springer, 1986, pp. 417–426.

[2] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[3] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[4] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Advances in Cryptology — CRYPTO 1984*, ser. Lecture Notes in Computer Science, vol. 196. Springer, 1985, pp. 10–18.

[5] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology — EUROCRYPT 2010*, ser. Lecture Notes in Computer Science, vol. 6110. Springer, 2010, pp. 1–23.

[6] C. Costello, P. Longa, and M. Naehrig, "Efficient algorithms for supersingular isogeny Diffie-Hellman," in *Advances in Cryptology — CRYPTO 2016*, ser. Lecture Notes in Computer Science, vol. 9814. Springer, 2016, pp. 572–601.

[7] C. P. L. Gouvêa and J. López, "Implementing GCM on ARMv8," in *Cryptographers' Track at the RSA Conference — CT-RSA 2015*, ser. Lecture Notes in Computer Science, vol. 9048. Springer, 2015, pp. 167–180.

[8] H. Seo, Z. Liu, Y. Nogami, J. Choi, and H. Kim, "Binary field multiplication on ARMv8," *Security and Communication Networks*, vol. 9, no. 13, pp. 2051–2058, 2016.

[9] H. Seo, "Faster ECC over $\mathbb{F}_{2^{571}}$ (feat. PMULL)," Cryptology ePrint Archive, Report 2015/745, 2015, http://eprint.iacr.org/2015/745.

[10] OpenSSL, "OpenSSL-1.1.0b," Available for download at https://www.openssl.org, Sep. 2016.

[11] National Institute of Standards and Technology (NIST), "Digital signature standard (DSS)," Federal Information Processing Standard, FIPS PUB 186-4, Jul. 2013.

[12] P. G. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM systems journal*, vol. 29, no. 4, pp. 526–538, 1990.

[13] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing elliptic curve cryptography and RSA on 8-bit CPUs," in *International Workshop on Cryptographic Hardware and Embedded Systems — CHES 2004*, ser. Lecture Notes in Computer Science, vol. 3156. Springer, 2004, pp. 119–132.

[14] M. Scott and P. Szczechowiak, "Optimizing multiprecision multiplication for public key cryptography," Cryptology ePrint Archive, Report 2007/299, 2007, http://eprint.iacr.org/2007/299.

[15] M. Hutter and E. Wenger, "Fast multi-precision multiplication for public-key cryptography on embedded microprocessors," in *International Workshop on Cryptographic Hardware and Embedded Systems — CHES 2011*, ser. Lecture Notes in Computer Science, vol. 6917. Springer, 2011, pp. 459–474.

[16] H. Seo and H. Kim, "Multi-precision multiplication for public-key cryptography on embedded microprocessors," in *Information Security Applications — WISA 2012*, ser. Lecture Notes in Computer Science, vol. 7690. Springer Verlag, 2012, pp. 55–67.

[17] Y. Lee, I.-H. Kim, and Y. Park, "Improved multi-precision squaring for low-end RISC microcontrollers," *Journal of Systems and Software*, vol. 86, no. 1, pp. 60–71, 2013.

[18] H. Seo, Z. Liu, J. Choi, and H. Kim, "Multi-precision squaring for public-key cryptography on embedded microprocessors," in *International Conference on Cryptology in India — INDOCRYPT 2013*, ser. Lecture Notes in Computer Science, vol. 8250. Springer, 2013, pp. 227–243.

[19] H. Seo, T. Park, S. Heo, G. Seo, B. Bae, L. Zhou, and H. Kim, "Multi-precision squaring for public-key cryptography on embedded microprocessors, a step forward," in *International Workshop on Information Security Applications*, 2016.

[20] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," in *Soviet Physics Doklady*, vol. 7, 1963, pp. 595–596.

[21] M. Scott, "Missing a trick: Karatsuba variations," Cryptology ePrint Archive, Report 2015/1247, 2015, http://eprint.iacr.org/2015/1247.

[22] D. J. Bernstein, "Curve25519: New Diffie-Hellman speed records," in *Public Key Cryptography — PKC 2006*, ser. Lecture Notes in Computer Science, vol. 3958. Springer, 2006, pp. 207–228.

APPENDIX

Algorithm 3 Assembly code for the 128-bit Karatsuba multiplication

Input: operand pointers (`x1` and `x2`)
Output: result pointer (`x0`)

```
1: LDP x4, x5, [x2]                    {loading}
2: LDP x2, x3, [x1]
3: MOV x1, #0
4: MUL x6, x2, x4                      {A_L · B_L low}
5: UMULH x7, x2, x4                    {A_L · B_L high}
6: MUL x8, x3, x5                      {A_H · B_H low}
7: UMULH x9, x3, x5                    {A_H · B_H high}
8: ADDS x10, x6, x8
9: ADCS x11, x7, x9
10: ADCS x12, x1, x1
11: ADDS x7, x7, x10
12: ADCS x8, x8, x11
13: ADCS x9, x9, x12
14: SUBS x2, x2, x3                    {absolute values}
15: SBCS x3, x3, x3
16: EOR x2, x2, x3
17: AND x3, x3, #1
18: ADD x2, x2, x3
19: SUBS x4, x4, x5
20: SBCS x5, x5, x5
21: EOR x4, x4, x5
22: AND x5, x5, #1
23: ADD x4, x4, x5
24: EOR x3, x3, x5                     {combining the signs}
25: SUB x3, x3, #1
26: MUL x10, x2, x4                    {A_D · B_D low}
27: UMULH x11, x2, x4                  {A_D · B_D high}
28: EOR x10, x10, x3                   {two's complement}
29: EOR x11, x11, x3
30: AND x4, x3, #1
31: ADDS x10, x10, x4
32: ADCS x11, x11, x1
33: ADCS x3, x3, x1
34: ADDS x7, x7, x10
35: ADCS x8, x8, x11
36: ADCS x9, x9, x3
37: STP x6, x7, [x0, #0]               {storing}
38: STP x8, x9, [x0, #16]
```