Optimized Karatsuba Squaring on 8-bit AVR Processors

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

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

Abstract. Multi-precision squaring is a crucial operation for implementation of Elliptic Curve Cryptography. Particularly, when it comes to embedded processors, the operation should be designed carefully to execute expensive ECC operation on resource constrained devices. In order to bridge the gap between high overheads and limited computation capabilities, we present optimized Karatsuba squaring method for embedded processors. Traditional squaring computation can be divided into two sub-squaring and one sub-multiplication parts. Firstly we compute the multiplication part with the fastest Karatsuba multiplication and then remaining two squaring parts are conducted with the fastest sliding block doubling squaring. Proposed method sets the new speed records for multi-precision squaring, improving the execution time by up to 8.49% comparing to the best known works.

Keywords: Multi-Precision Squaring, AVR, Karatsuba, Public Key Cryptography

1 Introduction

The developments of embedded processors make various technologies including Wireless Sensor Networks (WSNs) and Internet of Things (IoT) feasible. For the first step, the processors are deployed and continuously exchange the data what they obtained from the target areas. After then the data is re-produced into useful applications, i.e. home automation, smart grid and surveillance systems. However, WSNs and IoT are highly vulnerable to malicious attacks, because the data packets exchanged through the air can be easily captured by attackers. In order to provide trustworthy applications, users should construct secure and robust networking protocols. Under these circumstances, Public-Key Cryptography (PKC) would be a promising candidate that resolves vulnerabilities, since PKC applications including RSA [?], Elliptic Curve Cryptography (ECC) [?,?], provide quite useful protocols such as data encryption, key establishment as well as digital signature protocols. Among them, ECC is practical solution for resource constrained devices by considering the key size, which occupies small memory spaces rather than RSA. In order to establish ECC on embedded processor, we should design efficient cryptographic arithmetic operations over finite fields. Among these field arithmetic operations, multi-precision multiplication and squaring are crucial operations, thus, especially concerns on optimal implementation of these operations are deserved to pay for resource constrained environments. Recently high speed Karatsuba multiplication is introduced by Hutter and Peter in [?]. The work employs the subtractive Karatsuba multiplication to reduce the number of partial product computations by imposing simple addition operations. In case of squaring, Sliding Block Doubling (SBD) method is introduced by Seo et al. in INDOCRYPT'13 [?]. The method uses product-scanning for main computation and simply

⋆ Corresponding Author

doubles the duplicated parts to reduce the clock cycles into roughly half. However, there is no practical Karatsuba squaring results on AVR conducted. In this paper, we present Hybrid Karatsuba squaring on AVR, which finely combines Karatsuba multiplication and SBD squaring into single squaring operation. Firstly, we largely divided the squaring structure into two sub-squaring and one sub-multiplication. We adopt Karatsuba multiplication for sub-multiplication part and two remaining sub-squaring are established with SBD methods. When implementing on the ATmega128 microprocessor, only 3253 clock cycles are required for squaring at the length of 256-bit, which produces the fastest implementation published for 8-bit AVR platform, improving in a factor of 8.49%.

The rest of the paper is organized as follows. In Section 2, we describes the multi-precision multiplication and squaring algorithms. Proposed method is shown in Section 3. In Section 4, performance evaluation including clock cycles and memory consumptions are drawn. We conclude the paper in Section 5.

2 Field Multiplication and Squaring

Finite field multiplication and squaring are most expensive operations in ECC. In order to boost performance in speed factor, we should concern the finite field operations. We firstly explore multiplication methods. Our squaring method exploits the latest multiplication method.

2.1 Multi-precision Multiplication Techniques

Before describing the multi-precision multiplication method, we first define the following notations. Let A and B be two operands with a length of m-bit that are represented by multiple-word arrays. Each operand is written as follows: $A = (A[n-1], A[n-2], \ldots, A[1], A[0])$ and $B = (B[n-1], B[n-2], \ldots, B[1], B[0])$, whereby $n = \lceil m/w \rceil$, and w is the word size. The product of multiplication $A \cdot B$ is twice the length of A and can be represented by $C = (C[2n-1], C[2n-2], \ldots, C[1], C[0])$. For clarity, we describe the method using a multiplication structure and rhombus form. As shown in Figure 1, each point represents a word-level multiplication, i.e. $A[i] \times B[j]$. The rightmost corner of the rhombus represents the lowest index ($i, j = 0$), meanwhile the leftmost represents corner with highest index ($i, j = n-1$). The lowermost side represents result index $C[k]$, which ranges from the rightmost corner ($k = 0$) to the leftmost corner ($k = 2n-1$).

Operand Scanning Method Figure 1 shows the operand scanning which consists of two parts, i.e., inner and outer loops. In the inner loop, operand $A[i]$ holds a value and computes the partial product by multiplying all the multiplicands $B[j]$ ($j = 0...n-1$). While in the outer loop, the index of operand $A[i]$ increases by a word-size and then the inner loop is executed.

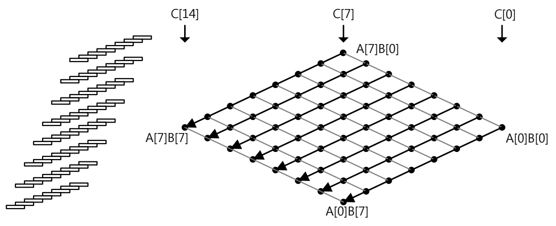

Fig. 1. Operand scanning multiplication [?]

Product Scanning Method Figure 2 shows the product scanning method which computes all partial products in the same column by multiplication and addition [?]. Since each partial product in the column is computed and then accumulated, registers are not needed for intermediate results. The results are stored once, and the stored results are not reloaded since all computations have already been completed.

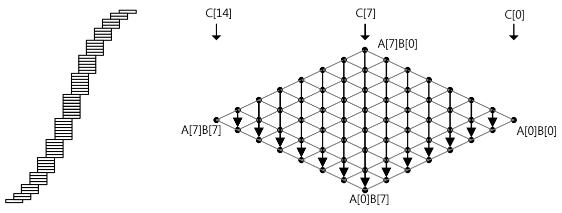

Fig. 2. Product scanning multiplication [?]

Hybrid Scanning Method Figure 3 shows the hybrid scanning method which combines both of the advantages of operand scanning and product scanning. Multiplication is performed on a block scale using product scanning. Inner block partial products follow the operand scanning rule. Therefore, this method reduces the number of `load` instructions by sharing the operands within the block [?].

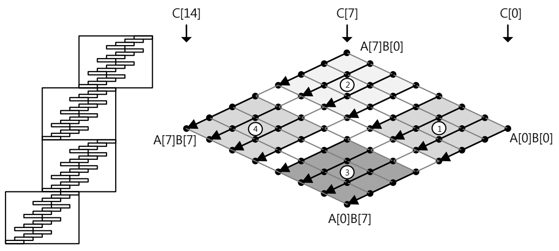

Fig. 3. Hybrid scanning multiplication [?]

Operand Caching Method Figure 4 shows the operand caching method which follows the product scanning method, but it divides the calculation into several row sections [?]. By reordering the sequence of inner and outer row sections, previously loaded operands in working registers are reused for the next partial products. A few `store` instructions are added, but the number of required `load` instructions is reduced. The number of row section is given by $r = \lfloor n/e \rfloor$, and e denotes the number of words used to cache digit in the operand.

Consecutive Operand Caching Method Figure 5 shows the consecutive operand caching which is based on characteristic of operand-caching method. Previous method has to reload operands whenever a row is changed which generates unnecessary overheads. To avoid these shortcomings, this method provides a contact point among rows that share the common operands for partial products. As a result of this, one side of operands is continuously maintained in registers and used [?].

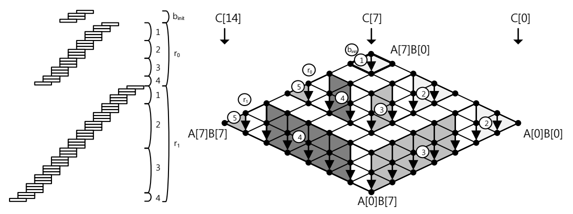

Fig. 4. Operand caching multiplication [?]

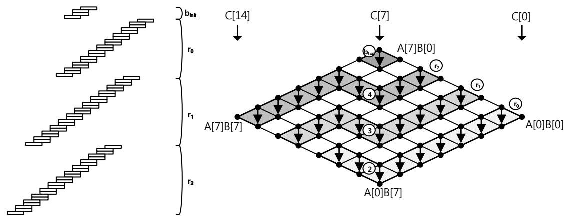

Fig. 5. Consecutive operand caching multiplication [?]

Subtractive Karatsuba Method The subtractive Karatsuba is established with following computations where the operands A and B has even m-bit variables and $k = m/2$.

- Write $A = A_{LOW} + 2^k A_{HIGH}$ and $B = B_{LOW} + 2^k B_{HIGH}$
- Compute $L = A_{LOW} \cdot B_{LOW}$
- Compute $H = A_{HIGH} \cdot B_{HIGH}$
- Compute $M = |A_{LOW} - A_{HIGH}| \cdot |B_{LOW} - B_{HIGH}|$
- Set $t = 0$, if $M = (A_{LOW} - A_{HIGH}) \cdot (B_{LOW} - B_{HIGH})$
- Otherwise $t = 1$
- Compute $M' = (-1)^t M$
- Obtain $AB = L + 2^k(L + H - M') + 2^m H$

This subtractive Karatsuba avoids the carry bits in operands ($|A_{LOW} - A_{HIGH}|$, $|B_{LOW} - B_{HIGH}|$) for the computation of middle partial products (M) but instead it needs to compute two absolute differences from $|A_{LOW} - A_{HIGH}|$ and $|B_{LOW} - B_{HIGH}|$ and one conditional negation of M. This has to be conducted in constant time computation to make the multiplication routine suitable for timing-attack-protected implementations of cryptographic primitives [?]. In order to ensure that, we need to choose branch-free operation rather than branched one. The branch-free method is established by masking operand, which outputs zeroed or original operand so without conditional branch we can get complete results.

2.2 Multi-precision Squaring Techniques

A typical software implementation of squaring method can be realized either using one of the above mentioned multiplication techniques or the specialized squaring method. Implementation using a specialized squaring method may have two advantages than simply using multiplication method for squaring. Firstly, in Figure 6, only one operand (A) is used for squaring computation, thus, the number of operand `load` is reduced to half times of multiplication, and many registers used for operand holding previously become idle status and can be used for caching intermediate results or other values. Second, there are duplicated partial products. In Figure 6, the squaring structure consists of three parts including red dotted middle part, light and dark gray triangle parts. The red part is multiplying a same operand, which is computed once. The other parts including light and dark gray parts generate same partial product results. For this reason, these parts are multiplied once and added twice to intermediate results. This computation generates same results, we expected. After removing duplicated partial product results, we can describe the squaring structure as a triangular form in Figure 7.

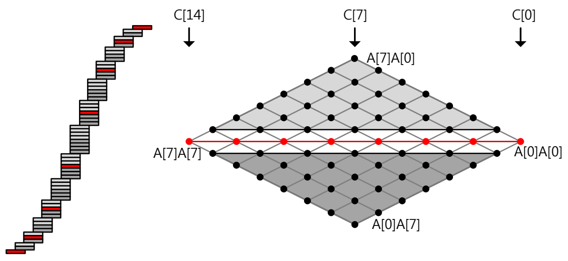

Fig. 6. Multi-precision squaring structure. Before removing duplicated partial product results.

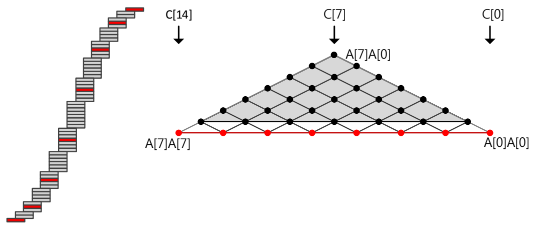

Fig. 7. Multi-precision squaring structure. After removing duplicated partial product results

Yang-Hseih-Lair Method Figure 8 describes Yang et al's method [**?**]. This squaring method is designed for hardware machine not for software implementation. First, duplicated partial products are computed using operand scanning. After then the intermediate results are doubled. Lastly, remaining partial products are computed.

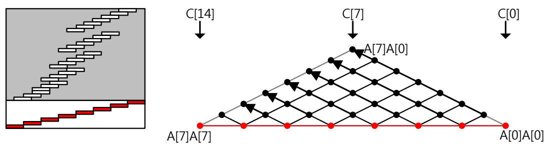

Fig. 8. Yang et al.'s squaring [**?**]

Carry Catcher Method Prime field multiplication or squaring consists of a number of partial products. When we compute partial products in ascending order, intermediate results generate carry values, accumulating the partial product results. In order to handle carry bit properly, carry-catcher method is introduced, which stores carry values to additional registers and then updates at the end of computation at once. In Figure 9, carry catcher based squaring was introduced by [**?**]. This method follows hybrid-scanning and doubles partial product results before they are added to results. This method is inefficient because all products should be doubled.

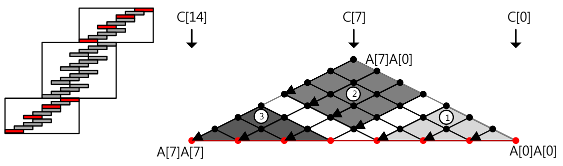

Fig. 9. Carry catcher squaring [?].

Lazy Doubling Method In Figure 10, efficient doubling method named lazy-doubling is described [?]. This method also follows hybrid scanning structure. The inner loop is computed in a operand scanning way, and then carry catcher method is used for removing consecutive carry updates. The strong feature of this method is doubling process which is delayed to end of inner structure and then computed. This method reduces number of arithmetic operations by conducting doubling computations on accumulated intermediate results. This technique significantly reduces a number of doubling process to one doubling process.

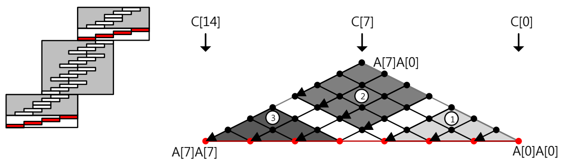

Fig. 10. Lazy doubling squaring [?]

Sliding Block Doubling Method In Figure 11, Sliding Block Doubling (SBD) method computes doubling using "1-bit left shifting" operation at the end of duplicated partial product computation, which accumulates all partial product results with only consuming few arithmetic operations. On the other hand, contrary to previously known solutions, SBD method adopts product-scanning method to compute duplicated product parts. After then the intermediate results are doubled to the final results.

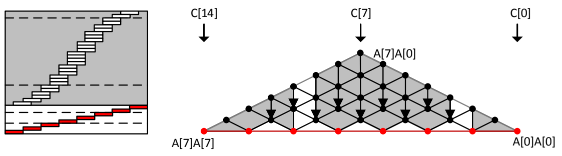

Fig. 11. Sliding block doubling squaring [?].

3 Hybrid Karatsuba Squaring

Previous squaring methods conduct single computation methods including operand-, product- or hybrid-scanning for partial product computations. However, we tried to use two different approaches into the squaring computation. As drawn in Figure 12, a squaring consists of three sub-structures including one sub-multiplication and two sub-squaring parts. The partial products ($A[4 \sim 7] \times A[0 \sim 3]$) construct sub-multiplication form with half size of operand

(4, $n/2$). Remaining partial products, ($A[0 \sim 3] \times A[0 \sim 3]$) and ($A[4 \sim 7] \times A[4 \sim 7]$) establish sub-squaring form with remaining half size of operand (4, $n/2$). Our main idea is finely dividing the squaring into sub-multiplication and sub-squaring structures. After then we apply different computation tactics to the different structures. When we use dedicated multiplication and squaring methods for multiplication and squaring, both operations are efficiently computable for each operation.

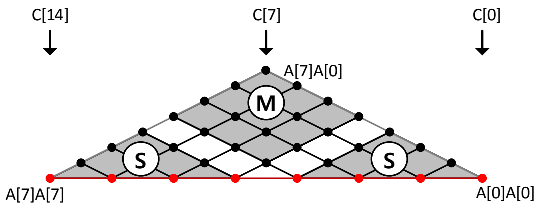

Fig. 12. Structure of squaring

In Figure 13, we describe a novel Hybrid Karatsuba squaring method. Firstly we divide ordinary squaring structure into three sub-parts. Part 1 colored in yellow is computed with best known Karatsuba multiplication method and remaining parts including 2 and 3 are established with SBD method. The part 1 is half size of operand (4, 8/2) and conducts subtractive Karatsuba multiplication for partial products ($A[4 \sim 7] \times A[0 \sim 3]$) and outputs intermediate results ($C[4] \sim C[11]$). Secondly parts 2 and 3 are computed with SBD method. Firstly in part 2, partial products from ($A[0] \times A[1]$) to ($A[3] \times A[2]$) are computed in product-scanning method, accumulating intermediate results ($C[4] \sim C[7]$). After then the intermediate results are doubled with one-bit left shift. Finally remaining partial products ($A[0] \times A[0] \sim A[3] \times A[3]$) are executed. In part 3, partial products from ($A[4] \times A[5]$) to ($A[7] \times A[6]$) are computed in product-scanning method, accumulating intermediate results ($C[8] \sim C[11]$). After then the intermediate results are doubled and remaining partial products ($A[4] \times A[4] \sim A[7] \times A[7]$) are executed. We implemented ECC friendly operand sizes including 128-, 160-, 192-, and 256-bit. The cases conduct 64-, 80-, 96-, and 128-bit subtractive karatsuba multiplication once for part 1 and then two 64-, 80-, 96-, and 128-bit SBD squaring operations are conducted for part 2 and 3, respectively. For SBD method, we combined three different operations (product-scanning, doubling, remaining partial product) into one computation process, because the target field size (64 ∼ 128-bit) needs small number of registers so they are readily retained into general purpose registers of target processor. This approach avoids intermediate results loading and storing overheads to/from memory.

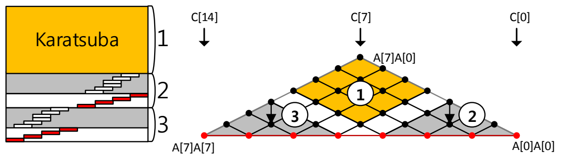

Fig. 13. Hybrid karatsuba squaring

4 Evaluation

In this section, we evaluate the performance of proposed squaring method in term of execution time and memory consumptions on 8-bit embedded platforms.

4.1 Evaluation on 8-bit Platform ATmega128

We implemented the method on 8-bit AVR processor ATmega128 which is widely used in MICAz mote, and then simulated our implementation over AVR Studio 6.0. Normally, an ATmega128 processor runs at a frequency of 7.3728 MHz. It has a 128 KB EEPROM chip and 4 KB RAM chip [?]. The ATmega128 processor also supports a RISC architecture with 32 registers, among which 6 registers serve as the special pointers for indirect addressing. The remaining 26 registers are available for arithmetic operations. One arithmetic instruction incurs 1 clock cycle, and memory addressing (e.g. `load`, `store`) or 8-bit multiplication (e.g. `mul`) incurs 2 processing cycles [?]. We used 4 registers for the operand and result pointers, 2 registers for storing the result of multiplication, and the remaining 26 registers for caching operands and intermediate results.

4.2 Constant-time solution: branched vesus branch-free

Our squaring implementation is hybrid approach combining both Karatsuba multiplication and SBD squaring. Particularly, Karatsuba multiplication is based on previous subtractive Karatsuba multiplication [?]. The subtractive Karatsuba multiplication has conditional branch to determine the sign-bit of middle partial product computations (M). In [?], they provide two different approaches including branched and branch-free to ensure constant time solution. First one is branched method, which uses conditional branch operation but it is also constant time method by adjusting the clock cycle with `NOP` (no operation). The alternative approach is branch-free which executes identical program code for both conditional branches. For this reason, the branch-free method is more secure than branched approach when it comes to side channel attack but it imposes additional computation costs to mask operands.

4.3 Karatsuba squaring on the AVR

Our 128-, 160- and 192-bit squaring are rooted from one level 64-, 80- and 96-bit Karatsuba multiplication. For the 64-bit multiplication ($k = 4$) we loaded all input bytes which need only $6k = 24$ registers. Afterwards we can also use the 4 registers holding the pointers to the inputs, i.e., X and Y, and thus have $7k = 28$ registers available. For the 80-bit multiplication, we spill one byte of the Z output address-pointer to/from the stack and to use it during the Karatsuba multiplication. The 96-bit multiplication needs all available registers including all 6 address-pointer bytes (X, Y, and Z) and we spilled two intermediate values in addition to the stack. For 80-bit and 96-bit Karatsuba multiplication we need more memory operations than strictly necessary to load inputs and store outputs. However, for those input sizes the benefit of smaller arithmetic complexity is so large that Karatsuba still clearly outperforms to the best known product-scanning. The 64-bit multiplication needs 16 `LD/LDD` instructions, and 16 `ST/STD` instructions. The 80-bit multiplication needs 25 `LD/LDD` instructions, and 20 `ST/STD` instructions. The 96-bit multiplication needs 42 `LD/LDD` instructions, 26 `ST/STD` instructions, 4 `PUSH` instructions, and 4 `POP` instructions. In terms of 256-bit squaring, we used two levels of Karatsuba multiplication in 128-bit. For 128-bit Karatsuba, we need $2k = 16$ registers for storing M and $k = 8$. For 2 levels Karatsuba multiplication we recursively conduct Karatsuba multiplication. In total, the 128-bit multiplication needs 92 `LD/LDD` instructions, 51 `ST/STD` instructions, 2 `PUSH` instructions, and 2 `POP` instructions. For remaining two squaring parts, we conduct 64-, 80-, 96- and 128-bit wise SBD methods twice. Firstly we loaded all operands (n) into general purpose register and 3 registers are assigned to result accumulation. We combined product-scanning, doubling and remaining parts in a process because AVR has ample storages to contain target operands. While SBD operation, we load intermediate results executed in former Karatsuba multiplication by n. The final results by $2n$ are stored into memory. This computation is conducted twice for two sub-squaring parts. In total, the two 64-bit squaring needs 32 `LD/LDD` instructions, 32 `ST/STD` instructions. The two 80-bit squaring needs 40 `LD/LDD` instructions, 40 `ST/STD` instructions. The two 96-bit squaring needs 48 `LD/LDD` instructions, 48 `ST/STD` instructions. The two 128-bit squaring needs 64 `LD/LDD` instructions, 64 `ST/STD` instructions.

4.4 Comparison results

In this section, we compared our Hybrid Karatsuba squaring with previous best known results. In Table 1, multiplication and squaring results are drawn. Compared with best known multiplication results, our squaring with branched and branch-free methods outperform 26% $\sim$ 32% and 25% $\sim$ 31% in speed factor, respectively. In terms of memory consumptions, branched and branch-free methods show 20% $\sim$ 28% and 20% $\sim$ 29% smaller size. This means that dedicated squaring method is far much beneficial than the best known multiplication methods when it comes to squaring. Compared with squaring method, our squaring with branched and branch-free methods outperform 2.8% $\sim$ 8.5% and 2.1% $\sim$ 7.3% in speed factor, respectively. In terms of memory consumptions, branched and branch-free methods show 0.8% $\sim$ 2.8% and 1.4% $\sim$ 3.1% smaller size. Generally, our branched method has higher speed than branch-free method but it needs more memory consumptions because branch-free only needs one program code for branch condition but branched method needs to write two different codes for both cases.

5 Conclusions

This paper presents a new technique of implementing multi-precision squaring on resource-constraint embedded processors, named Hybrid Karatsuba method. As the name implies, Hybrid Karatsuba method employs the Karatsuba multiplication and SBD squaring. Firstly squaring structure is finely divided into sub-multiplication and two sub-squaring parts and then sub-multiplication and remaining sub-squaring parts are conducted with Karatsuba multiplication and SBD squaring methods, respectively. In order to further achieve high speed footprint, we re-organize the computation order in efficient way. To validate the practical results, we implement the method on 8-bit AVR microcontroller with different length of operands. The results show that the method only requires 3253 clock cycles for performing a 256-bit squaring, achieving speed setting record on identical platform. Under a fair comparison, proposed method outperforms the best previous squaring work on same target device by a factor of 8.49%, depending on concrete bit-length. Moreover, proposed method can be easily extended for other embedded platforms with slight modification, e.g. 16-bit MSP and 32-bit ARM series processors. As a future work, we will apply our method to various platforms and show impact of proposed methods in real public key applications [**?**].

References

1. ARM. Cortex-A9 NEON Media Processing Engine Technical Reference Manual Revision: r4p1. Available for download at `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0409i/index.html`, June 2012.
2. P. D. Barrett. Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in Cryptology — CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer Verlag, 1987.

Table 1. Comparison of multiplication and squaring in clock cycle

		Input size (bits)			
Method		128	160	192	256
Multiplication					
Operand Caching [?]	cycles	n/a	2395	3469	6123
	bytes	n/a	3696	5354	9476
Consecutive operand caching [?]	cycles	1532	2356	3464	6180
	bytes	n/a	3652	n/a	n/a
(Consecutive) operand caching [?]	cycles	n/a	2346	3437	6115
	bytes	n/a	3632	n/a	n/a
Karatsuba(branched) [?]	cycles	1325	1998	2923	4797
	bytes	2228	3262	4602	8022
	stack	1	19	36	58
Karatsuba(branch-free) [?]	cycles	1369	2054	2987	4961
	bytes	2156	3150	4492	7614
	stack	1	19	36	58
Squaring					
Carry Catcher [?]	cycles	1365	2065	2909	5029
	bytes	n/a	n/a	n/a	n/a
Lazy Doubling [?]	cycles	1039	1509	2107	3559
	bytes	n/a	n/a	n/a	n/a
Sliding Block Doubling [?]	cycles	1003	1456	2014	3555
	bytes	1762	2378	3358	n/a
This paper(branched)	cycles	**974**	**1404**	**1966**	**3253**
	bytes	**1712**	**2376**	**3316**	**5442**
	stack	**0**	**0**	**4**	**1**
This paper(branch-free)	cycles	**982**	**1415**	**1982**	**3297**
	bytes	**1708**	**2374**	**3310**	**5370**
	stack	**0**	**0**	**4**	**1**

3. D. J. Bernstein and P. Schwabe. Neon crypto. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 320–339. Springer, 2012.
4. Bo Lin. Solving Sequential Problems in Parallel: An SIMD Solution to RSA Cryptography. Available for download at `http://cache.freescale.com/files/32bit/doc/app_note/AN3057.pdf`, Feb. 2006.
5. J. W. Bos and M. E. Kaihara. Montgomery multiplication on the cell. In *Parallel Processing and Applied Mathematics*, pages 477–485. Springer, 2010.
6. 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.
7. D. Câmara, C. P. Gouvêa, J. López, and R. Dahab. Fast software polynomial multiplication on arm processors using the neon engine. In *Security Engineering and Intelligence Informatics*, pages 137–154. Springer, 2013.
8. A. Faz-Hernandez, P. Longa, and A. H. Sanchez. Efficient and secure algorithms for glv-based scalar multiplication and their implementation on glv-gls curves. Technical report, Cryptology ePrint Archive, Report 2013/158, 2013. http://eprint.iacr.org, 2013.
9. Free Software Foundation, Inc. GMP: The GNU Multiple Precision Arithmetic Library. Available for download at `http://www.gmplib.org/`, Aug. 2014.
10. 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.
11. 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.
12. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.
13. 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.
14. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.
15. 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.
16. J.-J. Quisquater. Procédé de codage selon la méthode dite rsa, par un microcontrôleur et dispositifs utilisant ce procédé. *Demande de brevet français.(Dépôt numéro: 90 02274)*, 122, 1990.
17. J.-J. Quisquater. Encoding system according to the so-called rsa method, by means of a microcontroller and arrangement implementing this system, Nov. 24 1992. US Patent 5,166,978.
18. A. H. Sánchez and F. Rodríguez-Henríquez. Neon implementation of an attribute-based encryption scheme. Technical report, Technical Report CACR 2013-07, available at http://cacr.uwaterloo.ca/techreports/2013/cacr2013-07.pdf, 2013.
19. C. D. Walter and S. Thompson. Distinguishing exponent digits by observing modular subtractions. In *Topics in CryptologyCT-RSA 2001*, pages 192–207. Springer, 2001.
20. 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.

Appendix A. Algorithm for Hybrid Karatsuba Squaring Method

Appendix B. Example: Hybrid Karatsuba Squaring Structures for 128-, 160-, 192-, 256-bit Cases

Algorithm 1 Hybrid Karatsuba Squaring, where KM(A,B) and SBD(A) mean karatsuba multiplication on $A \times B$ and sliding block doubling on A^2, respectively

```
Require: word size $n$, Integers $A \in [0, n), C \in [0, 2n)$.
Ensure: $C = A^2$
 1: KM($A[0 \sim (n/2-1)], A[n/2 \sim (n-1)]$)
 2: SBD($A[0 \sim (n/2-1)]$) + $2 \times C[n/2 \sim (n-1)]$
 3: SBD($A[n/2 \sim n]$) + $2 \times C[n \sim (3n/2-1)]$
 4: return $C$
```

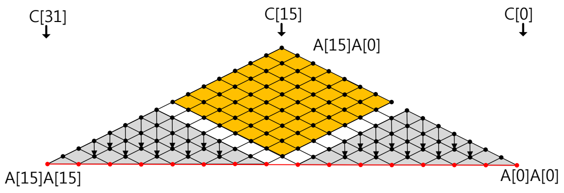

Fig. 14. Hybrid Karatsuba squaring for 128-bit

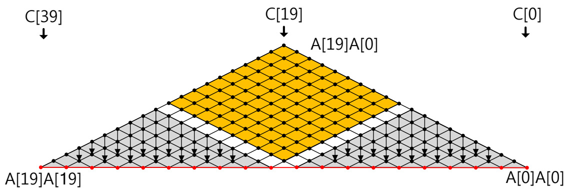

Fig. 15. Hybrid Karatsuba squaring for 160-bit

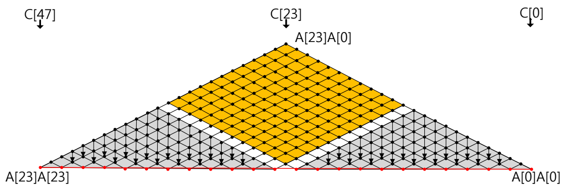

Fig. 16. Hybrid Karatsuba squaring for 192-bit

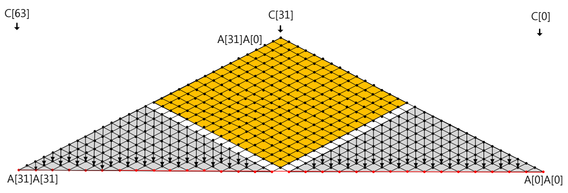

Fig. 17. Hybrid Karatsuba squaring for 256-bit