Faster ECC over $\mathbb{F}_{2^{521}-1}$ (feat. NEON)

Hwajeong Seo[1], Zhe Liu[2], Yasuyuki Nogami[3], Taehwan Park[1], Jongseok Choi[1], Lu Zhou[4], and Howon Kim[1(✉)]

[1] School of Computer Science and Engineering, Pusan National University, San-30, Jangjeon-Dong, Geumjeong-Gu, Busan 609–735, Republic of Korea
{hwajeong,pth5804,jschoi85,howonkim}@pusan.ac.kr
[2] Laboratory of Algorithmics, Cryptology and Security (LACS), University of Luxembourg, 6, rue R. Coudenhove-Kalergi, L-1359 Luxembourg-Kirchberg, Luxembourg
zhe.liu@uni.lu
[3] Graduate School of Natural Science and Technology, Okayama University, 3-1-1, Tsushima-naka, Kita, Okayama 700-8530, Japan
yasuyuki.nogami@okayama-u.ac.jp
[4] School of Computer Science and Technology, Shandong University, Jinan, China

Abstract. In this paper, we present high speed parallel multiplication and squaring algorithms for the Mersenne prime $2^{521} - 1$. We exploit 1-level Karatsuba method in order to provide asymptotically faster integer multiplication and fast reduction algorithms. With these optimization techniques, ECDH on NIST's (and SECG's) curve P-521 requires 8.1/4 M cycles on an ARM Cortex-A9/A15, respectively. As a comparison, on the same architecture, the latest OpenSSL 1.0.2d's ECDH speed test for curve P-521 requires 23.8/18.7 M cycles for ARM Cortex-A9/A15, respectively.

Keywords: Elliptic Curve Cryptography · P-521 · Karatsuba · SIMD · NEON

1 Introduction

Multi-precision modular multiplication and squaring are performance-critical building blocks of Elliptic Curve Cryptography (ECC). Since the algorithm is a computation-intensive operation, it demands careful optimizations to achieve

This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. 10043907, Development of high performance IoT device and Open Platform with Intelligent Software) and the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2015-H8501-15-1017) supervised by the IITP(Institute for Information & communications Technology Promotion).

S. Kwon and A. Yun (Eds.): ICISC 2015, LNCS 9558, pp. 169–181, 2016.
DOI: 10.1007/978-3-319-30840-1_11

acceptable performance particularly over embedded processors. 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 propagations [6]. The redundant representation reduces the number of active bits per register. Keeping the final result within remaining capacity of a register can avoid a number of carry propagations. In [2], vector instructions on the CELL microprocessor are used to perform multiplication on operands represented with a radix of 2^{16}. At CHES 2012, Bernstein and Schwabe adopted the reduced radix and presented an efficient modular multiplication on Curve25519. At HPEC 2013, a multiplicand reduction method in the reduced-radix representation was introduced for the NIST curves [8]. At CHES 2014, the Curve41417 implementation adopts 2-level Karatsuba multiplication in the redundant representation as well as a clever method to reduce inputs to the required multiplications rather than outputs [1]. Recently efficient Karatsuba multiplication algorithm for P-521 by Granger and Scott is proposed at PKC'15 which requires as few word-by-word multiplications as is needed for squaring, while incurring very little overhead from extra additions [4]. The algorithm shows high performance over 64-bit SISD architecture but it is not favorable for 32-bit ARM-NEON SIMD platforms because 32-bit SIMD architecture does not conduct 64-bit wise multiplication efficiently and needs to group the operands for parallel computations. Until now, there is relatively few studied on NIST's (and SECG's) curve P-521 for ARM-NEON architecture. Since the curve is NIST standard and ARM-NEON is the most well known smart phone processor, the efficient implementation of P-521 over ARM-NEON processor should be deserved. In this paper, we present speed record of P-521 over ARM-NEON platform. We exploit 1-level Karatsuba method in order to provide asymptotically faster integer multiplication and fast reduction algorithms.

The remainder of this paper is organized as follows. In Sect. 2, we recap the P-521 curve. In Sect. 3, we propose the efficient implementations of P-521 curve. In Sect. 4, we evaluate the performance of proposed methods in terms of clock cycles. Finally, Sect. 5 concludes the paper.

2 NIST Curve P-521

The Weierstrass form NIST curve P-521 as standardized in [3,7] and the finite field $\mathbb{F}_p$ is defined by:

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

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

$a =$ 01FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFC

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

and group order is defined by:

$$n = \texttt{01FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF}$$
$$\texttt{FFFFFFFF FFFFFFFA 51868783 BF2F966B 7FCC0148 F709A5D0 3BB5C9B8}$$
$$\texttt{899C47AE BB6FB71E 91386409}$$

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

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

For a point $P_2 = (X_2, Y_2, 1)$ which is affine point and not equal to P_1, let $P_3 = (X_3, Y_3, Z_3) = P_1 + P_2$. Then P_3 is computed as follows:

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

For a point P_2 which is projective point (X_2, Y_2, Z_2) and not equal to P_1, let $P_3 = (X_3, Y_3, Z_3) = P_1 + P_2$. Then P_3 is computed as follows:

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

3 Proposed Method

3.1 Multiplication

The prime of P-521 curve is $2^{521} - 1$. This representation can be written in $2^{522} - 2$ by following the idea of Langley in OpenSSL 1.0.0e approach. We choose 27/26-radix and this divides 522-bit into 20-limb as follows: (27, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26 ‖ 27, 26, 26, 26, 26, 26, 26, 26, 26, 26). Since the lower and

Algorithm 1. Karatsuba-based multiplication mod p_{521}

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

higher 261-bit wise operands share identical radix representation, we applied 1-level of Karatsuba multiplication which replaces the one 522-bit multiplication complexity to three 261-bit multiplications with some addition and subtraction operations. In this paper, we further improve the ordinary 1 level of Karatsuba multiplication particularly over $2^{522} - 2$. The detailed descriptions are available in Algorithm 1. For starter, both operands are divided into lower and higher parts from Steps $1 - 4$. In Step 5, lower part of operands are multiplied each other. In Step 6, higher parts are multiplied and then subtracted from the results of Step 5. Since the intermediate results exceed the 522-bit length, we directly reduce the intermediate results into range of modulus[1]. In Steps $7-9$, the results are divided into higher and lower parts and then lower parts are subtracted from higher parts. In Steps 10 and 11, higher and lower parts of operands are added each other. In Step 12, remaining several addition, subtraction and multiplication operations are conducted with direct reduction techniques. Finally, in Step 13, we obtained the results.

This approach introduces two advantages. First NEON engine over ARMv7 provides only 16 128-bit wise registers. For 261-bit multiplication, we need 5 registers for both 20-limb of 32-bit operands and 10 for 20-limb of 64-bit intermediate results and 1 for temporal registers. If we retain whole 522 multiplication results without reduction, the intermediate results exceed the size of general purpose registers, which introduces a number of memory load and store operations. Second this method follows basic concept of refined Karatsuba algorithm which reduces the one time of addition operation.

For 261-bit multiplication of 10-limb operand in (27, 26, 26, 26, 26, 26, 26, 26, 26, 26) representation, we can conduct multiplication as follows. The variables ($a_0 \sim a_9$ and $b_0 \sim b_9$) indicate the both operands and the other variables ($c_0 \sim c_{18}$) represents the intermediate results. The equation shows that some

[1] We discuss the detailed direct reduction techniques in following section.

of the partial product needs doubling the intermediate results to align the bit position. For example, the partial products including a_0b_2 and a_1b_1 are stored in same destination (c_2) but one (a_0b_2) is placed in 53-th and the other one (a_1b_1) is 52-th bit. In order to store the results in right bit position, we should conduct doubling on the partial product (a_1b_1) and get the doubled result ($2a_1b_1$) which is finally located in 53-th bit as like opponent (a_0b_2). The bit-aligned multiplication on 26/27 radix is as follows.

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

However, the aligned multiplication should be re-written in a SIMD friendly form. Particularly, the NEON architecture supports 2-way 32-bit wise multiplication, which means two consecutive 32-bit multiplications are computed and the store the two consecutive 64-bit results in one 128-bit register. For this reason, the alignments in the 128-bit register should be concerned in order to accumulate the multiplication results into correct destinations. We group the two adjacent partial product results as follows: $(c_1, c_0), (c_3, c_2)$, $(c_5, c_4), (c_7, c_6), (c_9, c_8)$, (c_{11}, c_{10}), $(c_{13}, c_{12}), (c_{15}, c_{14})$, (c_{17}, c_{16}). After then we re-arrange the partial products that need doubling the results to correct bit position. Since the doubling process can be computed together with multiplication by using instruction set (`vqdmull`), we can avoid one time of shift operation per each doubling operation. However, all partial products aren't grouped at once properly. We re-locate the intermediate results by conducting the shift to left by word size. This aligns the intermediate results as follows: $(c_2, c_1), (c_4, c_3)$, $(c_6, c_5), (c_8, c_7)$,

(c_{10}, c_9), (c_{12}, c_{11}), (c_{14}, c_{13}), (c_{16}, c_{15}), (c_{18}, c_{17}) and conducts the vectorized partial products.

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

In Step 6 of Algorithm 1, it conducts the partial product of $a_H \cdot b_H$ together with direct modular reduction. Since the reduction on our P-521 representation $(2^{522} - 2)$ only requires double addition/subtraction with values over 2^{522}, we conduct multiplication and double addition/subtraction with variables by calling `vqdmlal` and `vqdmlsl` instructions, respectively. The lower part of multiplication $(a_H \cdot b_H)$ is subtracted from intermediate results from c_{10} to c_{19}. Since the higher part of multiplication $(a_H \cdot b_H)$ is larger than modulus $(2^{522} - 2)$, we directly conduct reduction on intermediate results from c_0 to c_9. More in detail, firstly intermediate results are grouped in $(c_2, c_1), (c_4, c_3), (c_6, c_5), (c_8, c_7)$, $(c_{10}, c_9), (c_{12}, c_{11}),\ (c_{14}, c_{13}), (c_{16}, c_{15}), (c_{18}, c_{17})$. After then the lower parts of multiplication $(a_H \cdot b_H)$ are subtracted from intermediate results $(c_{11} \sim c_{19})$. The higher parts of multiplication $(a_H \cdot b_H)$ are directly subtracted from intermediate results $(c_0 \sim c_9)$. After then intermediate results are shift to right by word size and then conduct the remaining partial products in following group order: $(c_1, c_0), (c_3, c_2), (c_5, c_4), (c_7, c_6),\ (c_9, c_8), (c_{11}, c_{10}), (c_{13}, c_{12}), (c_{15}, c_{14}),\ (c_{17}, c_{16})$.

$$
\begin{aligned}
(a_9 \sim a_0) &\leftarrow (a_{19} \sim a_{10})\\
(b_9 \sim b_0) &\leftarrow (b_{19} \sim b_{10})\\
(c_{12}, c_{11}) &\leftarrow (c_{12}, c_{11}) - (2a_1b_1, a_1b_0)
\end{aligned}
$$

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

3.2 Squaring

Multi-precision squaring can be utilized with ordinary multiplication methods. However, squaring method has two advantages over the multiplication methods. Both partial products $A[i] \times A[j]$ and $A[j] \times A[i]$ output the identical results. By taking accounts of these features, the parts are multiplied with doubled form (i.e. $2 \times A[i] \times A[j]$) which provides the same results of conventional multiplication (i.e. $A[i] \times A[j] + A[j] \times A[i]$). We applied squaring on 261-bit wise operand as follows. Unlike multiplication operation, squaring can eliminate the almost half of partial product with doubling but this introduces quadrupled results. In order to resolve this matter, we firstly doubled the operands and preserve both original and doubled operands in the registers. This is possible approach, because squaring only needs one operand and remaining registers can retain the doubled operands. This ensures the quadrupled multiplication with doubled operands and double multiplication instruction such as `vqdmull`.

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

Similar with SIMD multiplication, squaring operation also needs to group the two intermediate results in SIMD friendly way. Squaring has one more advantage over that of multiplication. The whole squaring operation can be executed in following representation (c_1, c_0), (c_3, c_2), (c_5, c_4),(c_7, c_6), (c_9, c_8), (c_{11}, c_{10}), (c_{13}, c_{12}),(c_{15}, c_{14}), (c_{17}, c_{16}). Since squaring reduces the duplicated partial products, single group representation can cover the whole partial products without re-arrangements.

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

$$
\begin{aligned}
(c_{17}, c_{16}) &\leftarrow 2(a_8 a_9, a_7 a_9) \\
(t_1, t_0) &\leftarrow (a_6 a_6, 2a_5 a_5) \\
(t_3, t_2) &\leftarrow (a_8 a_8, a_7 a_7) \\
t_4 &\leftarrow a_9 a_9 \\
c_{10} &\leftarrow c_{10} + t_0 \\
c_{12} &\leftarrow c_{12} + t_1 \\
c_{14} &\leftarrow c_{14} + t_2 \\
c_{16} &\leftarrow c_{16} + t_3 \\
c_{18} &\leftarrow c_{18} + t_4
\end{aligned}
$$

We also applied the direct reduction techniques described in Steps 6 and 12 in Algorithm 1 for the squaring method as well. Firstly intermediate results are grouped in $(c_1, c_0), (c_3, c_2), (c_5, c_4), (c_7, c_6), (c_9, c_8), (c_{11}, c_{10}), (c_{13}, c_{12}), (c_{15}, c_{14}), (c_{17}, c_{16})$. After then the lower part of multiplication $(a_H \cdot a_H)$ is subtracted from intermediate results $(c_{11} \sim c_{19})$. The higher part of multiplication $(a_H \cdot a_H)$ is directly subtracted from intermediate results $(c_0 \sim c_9)$.

$$
\begin{aligned}
(a_9 \sim a_0) &\leftarrow (a_{19} \sim a_{10}) \\
(c_{11}, c_{10}) &\leftarrow (c_{11}, c_{10}) - (2a_0 a_1, a_0 a_0) \\
(c_{13}, c_{12}) &\leftarrow (c_{13}, c_{12}) - 2(a_0 a_3, a_0 a_2) - (4a_1 a_2, 2a_1 a_1) \\
(c_{15}, c_{14}) &\leftarrow (c_{15}, c_{14}) - 2(a_0 a_5, a_0 a_4) - 4(a_1 a_4, a_1 a_3) - (4a_2 a_3, 2a_2 a_2) \\
(c_{17}, c_{16}) &\leftarrow (c_{17}, c_{16}) - 2(a_0 a_7, a_0 a_6) - 4(a_1 a_6, a_1 a_5) - 4(a_2 a_5, a_2 a_4) - (4a_3 a_4, 2a_3 a_3) \\
(c_{19}, c_{18}) &\leftarrow (c_{19}, c_{18}) - 2(a_0 a_9, a_0 a_8) - 4(a_1 a_8, a_1 a_7) - \\
&\quad 4(a_2 a_7, a_2 a_6) - 4(a_3 a_6, a_3 a_5) - (4a_4 a_5, 2a_4 a_4) \\
(c_1, c_0) &\leftarrow (c_1, c_0) - (4a_2 a_9, 8a_1 a_9) - (4a_3 a_8, 8a_2 a_8) - (4a_4 a_7, 8a_3 a_7) - (4a_5 a_6, 8a_4 a_6) \\
(c_3, c_2) &\leftarrow (c_3, c_2) - 4(a_4 a_9, a_3 a_9) - 4(a_5 a_8, a_4 a_8) - 4(a_6 a_7, a_5 a_7) \\
(c_5, c_4) &\leftarrow (c_5, c_4) - 4(a_6 a_9, a_5 a_9) - 4(a_7 a_8, a_6 a_8) \\
(c_7, c_6) &\leftarrow (c_7, c_6) - 4(a_8 a_9, a_7 a_9) \\
(t_1, t_0) &\leftarrow (2a_6 a_6, 4a_5 a_5) \\
(t_3, t_2) &\leftarrow 2(a_8 a_8, a_7 a_7) \\
t_4 &\leftarrow 2a_9 a_9 \\
c_0 &\leftarrow c_{10} - t_0 \\
c_2 &\leftarrow c_{12} - t_1 \\
c_4 &\leftarrow c_{14} - t_2 \\
c_6 &\leftarrow c_{16} - t_3 \\
c_8 &\leftarrow c_{18} - t_4
\end{aligned}
$$

3.3 Inversion

Constant-time inversion is performed by powering by $p_{521} - 2 = 2^{521} - 3$. The inverse can be computed at a cost of 520S + 13M by following Algorithm 2.

Algorithm 2. Fermat-based inversion mod p_{521}

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

1: $a_2 \leftarrow a_1^2 \cdot a_1$ {cost: 1S+1M}
2: $a_3 \leftarrow a_2^2 \cdot a_1$ {cost: 1S+1M}
3: $a_6 \leftarrow a_3^{2^3} \cdot a_3$ {cost: 3S+1M}
4: $a_7 \leftarrow a_6^2 \cdot a_1$ {cost: 1S+1M}
5: $a_8 \leftarrow a_7^2 \cdot a_1$ {cost: 1S+1M}
6: $a_{16} \leftarrow a_8^{2^8} \cdot a_8$ {cost: 8S+1M}
7: $a_{32} \leftarrow a_{16}^{2^{16}} \cdot a_{16}$ {cost: 16S+1M}
8: $a_{64} \leftarrow a_{32}^{2^{32}} \cdot a_{32}$ {cost: 32S+1M}
9: $a_{128} \leftarrow a_{64}^{2^{64}} \cdot a_{64}$ {cost: 64S+1M}
10: $a_{256} \leftarrow a_{128}^{2^{128}} \cdot a_{128}$ {cost: 128S+1M}
11: $a_{512} \leftarrow a_{256}^{2^{256}} \cdot a_{256}$ {cost: 256S+1M}
12: $a_{519} \leftarrow a_{512}^{2^7} \cdot a_7$ {cost: 7S+1M}
13: $a_1^{2^{521}-3} \leftarrow a_{519}^{2^2} \cdot a_1$ {cost: 2S+1M}
14: **return** $a_1^{2^{521}-3}$

3.4 Addition and Subtraction

Addition and subtraction over redundant representations do not introduce the carry or borrow propagations from least significant word to most significant word. Since SIMD instruction conducts the four different addition or subtraction operations with single instruction, we conduct 20 26/27-radix addition/subtraction with five times of 32-bit wise vector addition/subtraction operations. For point addition and doubling, several addition variants such as integer doubling, tripling, quadrupling, octupling are required. We also exploit the vector addition by 1, 2, 2, 3 times for doubling, tripling, quadrupling and octupling operations, respectively. Since the tripling, quadrupling and octupling operations may generate the overflows in very next step, we conduct reduction right after the operations.

3.5 Radix Adjustments

The multiplication and squaring computations produce a product of the 63-bit 20 limbs for intermediate results. We then use a sequence of carries to bring each limb down to 26 or 27 bits. We vectorized between a carry $c_0 \rightarrow c_1$ and $c_{10} \rightarrow c_{11}$, between a carry $c_1 \rightarrow c_2$ and $c_{11} \rightarrow c_{12}$. The computation order is as follows: $(c_{10}, c_0) \rightarrow (c_{11}, c_1)$, $(c_{11}, c_1) \rightarrow (c_{12}, c_2)$, $(c_{12}, c_2) \rightarrow (c_{13}, c_3)$, $(c_{13}, c_3) \rightarrow (c_{14}, c_4)$, $(c_{14}, c_4) \rightarrow (c_{15}, c_5)$, $(c_{15}, c_5) \rightarrow (c_{16}, c_6)$, $(c_{16}, c_6) \rightarrow (c_{17}, c_7)$, $(c_{17}, c_7) \rightarrow (c_{18}, c_8)$, $(c_{18}, c_8) \rightarrow (c_{19}, c_9)$, $(c_{19}, c_9) \rightarrow (c_0, c_{10})$, $(c_{10}, c_0) \rightarrow (c_{11}, c_1)$. The computations output 20 limbs of results (27, 27, 26, 26, 26, 26, 26, 26, 26, 26 || 27, 27, 26, 26, 26, 26, 26, 26, 26, 26) (Table 1).

The addition and subtraction computations carry out the 31-bit wise 20 limbs. Similarly, we use a sequence of carries to bring each limb down to 26 or

27 bits. The computation order is as follows: $(c_{19}, c_9) \rightarrow (c_0, c_{10})$, $(c_{10}, c_0) \rightarrow (c_{11}, c_1)$, $(c_{11}, c_1) \rightarrow (c_{12}, c_2)$, $(c_{12}, c_2) \rightarrow (c_{13}, c_3)$, $(c_{13}, c_3) \rightarrow (c_{14}, c_4)$, $(c_{14}, c_4) \rightarrow (c_{15}, c_5)$, $(c_{15}, c_5) \rightarrow (c_{16}, c_6)$, $(c_{16}, c_6) \rightarrow (c_{17}, c_7)$, $(c_{17}, c_7) \rightarrow (c_{18}, c_8)$, $(c_{18}, c_8) \rightarrow (c_{19}, c_9)$. This computation outputs 20 limbs as follows (27, 26, 26, 26, 26, 26, 26, 26, 26, 27 || 27, 26, 26, 26, 26, 26, 26, 26, 26, 27). Unlike multiplication case, we firstly conduct the radix adjustment on the most significant group (c_{19}, c_9) which can reduce the one time of adjustment.

Table 1. Prime-field ECC timings from `openssl speed ecdh` on Cortex-A9 and Cortex-A15 devices where Cortex-A9 with OpenSSL 1.0.2d on a Odroid-X2 development board running at 1.7 GHz and Cortex-A15 with OpenSSL 1.0.2d on a Odroid-XU development board running at 1.6 GHz

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

3.6 Scalar Multiplication

Constant time scalar multiplication is computed with the window method. This consists of pre-computation of point and scalar multiplication by window width. For unknown point, we tested over three different window sizes including 4, 5 and 6. For window size 4, pre-computation needs 1 time of doubling and 7 times of addition. For window size 5, pre-computation needs 1 time of doubling and 15 times of addition. For window size 6, pre-computation needs 1 time of doubling and 31 times of addition. Without point pre-computation, scalar multiplication needs (131A+520D), (105A+520D), (87A+516D) for 4, 5, 6 window methods. Total (138A+521D), (120A+521D), (118A+517D) are needed for 4, 5, 6 window methods. For fixed point we conduct the comb window method. Since fixed point can take advantages of online pre-computation which reduces the number of point doubling, total overheads for 4, 5, 6 window methods are calculated in (131A+130D), (105A+104D), (87A+86D), respectively.

4 Evaluation

There are several works done over lower security levels including Curve41417 and Ed448-Goldilocks [1,5]. However it is hard to retrieve the fair performance evaluations due to different parameters. One obvious difference is that smaller curve

Table 2. Clock cycles for scalar multiplication

Target	Unknown point			Fixed point			ECDH
	w=4	w=5	w=6	w=4	w=5	w=6	
Cortex-A9	6,291,936	6,098,946	6,011,768	3,056,410	2,527,714	2,147,404	8,159,172
Cortex-A15	3,097,904	3,003,728	2,970,976	1,503,661	1,243,027	1,056,902	4,027,878

Table 3. Clock cycles for finite field multiplication, squaring and inversion; point addition and doubling

Target	Finite field arithmetic			Point operation	
	MUL	SQR	INV	ADD	DBL
Cortex-A9	708	578	311,451	12,453	8,036
Cortex-A15	350	276	149,208	6,176	3,962

only requires small number of general purpose registers which utilizes the more number of temporal registers than longer curve. Furthermore target modulus prime is different to each other which introduces totally different radix representations and fast reduction algorithms. For this reason, we evaluate the obvious candidate, latest OpenSSL 1.0.2d implementations using the command `openssl speed ecdh`. On the same architecture, OpenSSL 1.0.2d reports 71.2 and 85.3 operations per second for A9 and A15, which implies a count of approximately 23.8 M and 18.7 M cycles per ECDH. On the other hand, our implementations described in Table 2 only require 8.1 M and 4.0 M cycles per ECDH for A9 and A15, respectively. In Table 3, the detailed clock cycles for basic operations are drawn where the clock cycles of finite field operations include reduction operation and the point addition and doubling is calculated over Jacobian representations.

5 Conclusion

In this paper, we show efficient implementations of P-521 over ARM-NEON processor. We conduct 1-level of Karatsuba multiplication together with direct modular reduction on $(2^{522} - 2)$. By taking advantages of several optimization techniques, we improve the modular multiplication on P-521 significantly. Same technique is also applied to squaring and reduces the complexities in similar manner. Finally, we outperform the latest OpenSSL 1.0.2d over both A9 and A15 ARM processors.

References

1. Bernstein, D.J., Chuengsatiansup, C., Lange, T.: Curve41417: karatsuba revisited. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 316–334. Springer, Heidelberg (2014)

2. Bos, J.W., Kaihara, M.E.: Montgomery multiplication on the cell. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part I. LNCS, vol. 6067, pp. 477–485. Springer, Heidelberg (2010)
3. Standard for Efficient Cryptography Group: Recommended elliptic curve domain parameters (2000)
4. Granger, R., Scott, M.: Faster ECC over $\mathbb{F}_{2^{521}-1}$. In: Katz, J. (ed.) PKC 2015. LNCS, vol. 9020, pp. 539–553. Springer, Heidelberg (2015)
5. Hamburg, M.: Ed448-goldilocks, a new elliptic curve
6. Intel Corporation.: Using streaming SIMD extensions (SSE2) to perform big multiplications, Application note AP-941 (2000). http://software.intel.com/sites/default/files/14/4f/24960
7. U.D. of Commerce/N.I.S.T. Federal information processing standards publication 186–2 fipps 186–2 digital signature standard
8. Pabbuleti, K.C., Mane, D.H., Desai, A., Albert, C., Schaumont, P.: SIMD acceleration of modular arithmetic on contemporary embedded platforms. In: High Performance Extreme Computing Conference (HPEC), pp. 1–6. IEEE (2013)