High-Throughput Elliptic Curve Cryptography using AVX2 Vector Instructions

Hao Cheng, Johann Großschädl, Jiaqi Tian, Peter B. Rønne, and Peter Y. A. Ryan

DCS and SnT, University of Luxembourg
6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
{hao.cheng,johann.groszschaedl,peter.roenne,peter.ryan}@uni.lu
jiaqi.tian.002@student.uni.lu

Abstract. Single-Instruction-Multiple-Data (SIMD) extensions like Intel's AVX2 offer a great potential to accelerate elliptic curve cryptography compared to a straightforward implementation using only base x64 instructions. All existing AVX2 implementations of scalar multiplication on Curve25519 and alternative elliptic curves are optimized for low latency. We argue in this paper that many applications, most notably server-side TLS handshake processing, would benefit more from throughput-optimized implementations than latency-optimized ones. To support this argument we introduce throughput-optimized AVX2 implementations of variable-base scalar multiplication on Curve25519 and fixed-base scalar multiplication on Ed25519. Both implementations perform four scalar multiplications in parallel, whereby each scalar multiplication uses a 64-bit element of a 256-bit AVX2 vector. The field arithmetic is based on a radix-2^{29} representation of the field elements, which makes it possible to execute four parallel multiplications modulo a multiple of $p = 2^{255} - 19$ in just 88 Skylake cycles. Four variable-base scalar multiplications on Curve25519 require less than 250,000 Skylake cycles, which translates into a throughput of 32,318 scalar multiplications per second at a clock frequency of 2 GHz. For comparison, the currently best latency-optimized AVX2 implementation reaches a throughput of only about 21,000 scalar multiplications per second on the same Skylake processor.

Keywords: Throughput-optimized cryptography · Curve25519 · Single instruction multiple data (SIMD) · Advanced vector extensions (AVX2)

1 Introduction

Essentially any modern high-performance processor architecture supports vector instruction set extensions to enable parallel processing based on the Single Instruction Multiple Data (SIMD) paradigm. Typical and well-known examples of vector extensions include MMX, SSE, and AVX developed by Intel, AMD's 3DNow, and the AltiVec instruction set for the PowerPC. Besides architectures that target the personal computing and server markets, vector extensions have

also been integrated into instruction sets aimed at the embedded and mobile domain, e.g. ARM NEON. Taking Intel's x86/x64 platform as a case study, the evolution of vector extensions over the past 25 years can be briefly summarized as follows. In 1997, Intel introduced the MMX extensions for the 32-bit x86 architecture, which initially supported operations on packed integers using the eight 64-bit wide registers of the Floating-Point (FP) unit. Two years later, in 1999, Intel announced SSE, the first of a series of so-called Streaming SIMD eXtensions, enriching x86 by eight 128-bit registers (`XMM0` to `XMM7`) and dozens of new instructions to perform packed integer and FP arithmetic. Starting with the Sandy Bridge microarchitecture (released in early 2011), Intel equipped its x64 processors with AVX (Advanced Vector eXtensions), which added packed FP instructions using sixteen 256-bit registers (`YMM0` to `YMM15`). These registers are organized in two 128-bit lanes, whereby the lower lanes are shared with the corresponding 128-bit `XMM` registers. AVX2 appeared with Haswell in 2013 and enhanced AVX to support new integer instructions that are able to operate on e.g. eight 32-bit elements, four 64-bit elements or sixteen 16-bit elements in parallel. The most recent incarnation of AVX is AVX512, which augments the execution environment of x64 by 32 registers of a length of 512 bits and various new instructions. Consequently, the bitlength of SIMD registers increased from 64 to 512 over a period of just 20 years, and one can expect further extensions in the future. For example, as recently reported in [10], the RISC-V architecture will support vectors of a length of up to 16,384 bits.

Though originally designed to speed up audio/video processing and computer gaming, SIMD instructions sets like SSE, AVX or NEON turned out to be also beneficial for various kinds of cryptographic algorithms. Using prime-field-based Elliptic Curve Cryptography (ECC) as example, an implementer can take advantage of SIMD parallelism to accelerate (i) the field arithmetic by adding or multiplying several limbs of field elements in parallel, (ii) the curve arithmetic by executing e.g. two or four field operations in parallel, and (iii) a combination of both. The latency of arithmetic operations in a prime field $\mathbb{F}_p$ can be reduced with the help of SIMD instructions in a very similar way as described in e.g. [4, 8, 9] for other public-key cryptosystems, most notably RSA. All these implementations have in common that they employ the product-scanning technique [11] in combination with a reduced-radix representation (e.g. $w = 28$ bits per limb) to perform multi-precision multiplication in a 2-way parallel fashion, which means two $(w \times w \rightarrow 2w)$-bit multiplications are carried out simultaneously. Also the point arithmetic offers various possibilities for parallel execution. For example, the so-called ladder-step operation of the Montgomery ladder for Montgomery curves [14] can be implemented in a 2-way or 4-way parallel fashion so that two or four field operations are carried out in parallel, as described in e.g. [6, Algorithm 1] and [12, Fig. 1] for AVX2. The scalar multiplication on (twisted) Edwards curves [2] can be accelerated by parallel execution at the layer of the point arithmetic as well. For example, 2-way and 4-way parallel implementations of the point addition and point doubling were presented in e.g. [3, 5, 7] and [7], respectively; these execute either two or four field-arithmetic operations in par-

allel. Finally, there exist also implementations that combine parallelism at the field-arithmetic and point-arithmetic layer, which we characterize as $(n \times m)$-way parallel implementations: they perform n field operations in parallel, whereby each field operation is executed in an m-way parallel fashion and uses m elements of a vector. For example, Faz-Hernández et al. describe in [7] a (2×2)-way parallel AVX2 implementation of variable-base scalar multiplication on Curve25519 that executes in 121,000 Haswell cycles or 99,400 Skylake cycles. More recently, Hisil et al. [12] presented an AVX512 implementation of Curve25519 that is (4×2)-way parallelized (i.e. four field operations in parallel, each using two 64-bit elements) and achieved an execution time of 74,368 Skylake cycles.

Benchmarking results published in the literature show that parallel implementations of Curve25519 do not scale very well when switching from one generation of AVX to the next. While AVX512 (in theory) doubles the amount of parallelism compared to AVX2 (since it is capable to perform operations on eight 64-bit elements instead of four), the actual reduction in execution time (i.e. latency) is much smaller, namely just around 25% (74,368 vs. 99,400 Skylake cycles [12]). This immediately raises the question of how implementers can exploit the massive parallelism of future SIMD extensions operating on vectors that may be 1024 bits long, or even longer, given the diminishing gain achieved by Hisil et al [12]. Going along with this "How" question is the "Why" question, i.e. why are fast implementations of e.g. Curve25519 needed, or, put differently, what applications demand low-latency implementations of Curve25519. Unfortunately, none of the papers mentioned in the previous paragraph identifies a use case or a target application for their latency-optimized implementations. Since many security protocols nowadays support Curve25519 (e.g. TLS 1.3), one may argue that a fast implementation of Curve25519 reduces the overall handshake-latency a TLS client experiences when connecting to a TLS server. The problem with this reasoning is that transmitting the public keys over the Internet will most likely introduce an orders-of-magnitude higher latency than the computation of the shared secret. Furthermore, given clock frequencies of around 4 GHz, a user will most likely not recognize an execution-time reduction by a few 10,000 clock cycles. It could now be argued that variable-base scalar multiplication is not only needed on the client side, but has to be performed also by the TLS server. Indeed, TLS servers of big organizations like Google or Facebook may be confronted with several 10,000 TLS handshakes per second, and a faster Curve25519 implementation will help them to cope with such extreme workloads. However, what really counts on the server side is not the latency of a single scalar multiplication, but the throughput, i.e. how many scalar multiplications can be computed per second. Given this requirement, would it not make sense to optimize a Curve25519 implementation for high throughput instead of low latency? What throughput can a throughput-optimized implementation achieve compared to a latency-optimized implementation? To our surprise, it seems that both questions are not answered by the existing literature.

This paper takes a first step to answer these questions and introduces a throughput-optimized AVX2 implementation of variable-base scalar multiplica-

tion on Curve25519 and fixed-base scalar multiplication on Ed25519. Our implementation performs a (4×1)-way parallel scalar multiplication, which means we execute four scalar multiplication simultaneously in a SIMD fashion, whereby each can use a different scalar and, in the case of Curve25519, a different base point. Both the point arithmetic and the underlying field arithmetic of each scalar multiplication use only a single 64-bit element of a 256-bit AVX2 vector. This "coarse-grained" form of parallelism has the advantage that it is easy to implement (by simply vectorizing a reduced-radix implementation for a 32-bit platform) and easy to audit. In contrast to most previous AVX2 implementation, we employ a radix-2^{29} representation of the field elements (i.e. 29 bits per limb), which turned out to be the best option for our (4×1)-way scalar multiplication when we analyzed various alternative approaches (including the classical 25/26-bits per limb variant [1]). Our benchmarking results show that, on a Skylake processor, four scalar multiplications can be performed in less than 250,000 clock cycles. For comparison, the currently best latency-optimized AVX2 implementation needs more than 374,000 Skylake cycles to execute four variable-base scalar multiplications on Curve25519, which means our software achieves a 1.5 times higher throughput than the current leader in the low-latency domain.

Availability of the Software. The source code of our software is publicly available at `https://gitlab.uni.lu/APSIA/AVXECC` and released under GPLv3 license.

2 AVX2 Instruction Set

Advanced Vector eXtension version 2 (AVX2) is an instruction set extension for SIMD enhancement, supporting packed-integer operations using 256-bit wide registers. It was announced by Intel in 2011 and first supported with Intel's Haswell microarchitecture in 2013, which is thus also known as the Haswell New Instructions. Since then, all the subsequent Intel Core microarchitectures inherited the AVX2 unit, e.g. Broadwell, Skylake, Icelake and etc. We carried out our experiments on the Haswell and Skylake microarchitectures since these both are often used as the reference platforms in related works such as [7, 12, 15]. On both Haswell and Skylake microarchitectures, instructions (including AVX2 instructions) are fetched from the instruction cache and then decoded into micro-operations (micro-ops) by the *front end*. Afterwards, micro-ops are stored in a pool and will be assigned by the *superscalar execution engine* to available execution *ports*. Specifically, the execution engine deals with micro-ops in a so-called *out-of-order* way, i.e. the execution port can handle a micro-op before another micro-op that is from an earlier-decoded instruction. This feature allows the processor to deal with the later instructions whose operands are ready and improves CPU processing efficiency. There are totally eight execution ports with different functionalities on both Haswell and Skylake microarchitectures, namely ports 0 to 7. For both microarchitectures, ports 0, 1 and 5 can handle micro-ops of vector instructions; ports 2, 3, 4 and 7 deal with memory access;

while port 6 is able to process branchings. As for AVX2-related ports (ports 0, 1 and 5), they have different processing abilities on Haswell and Skylake. In detail, some primary AVX2 instructions with the corresponding execution ports are listed in the following:

- `VPMULUDQ`: port 0 on Haswell; ports 0 and 1 on Skylake.
- `VPADDQ`: ports 1 and 5 on Haswell; ports 0, 1 and 5 on Skylake.
- `VPAND`: ports 0, 1 and 5 on Haswell; ports 0, 1 and 5 on Skylake.

For instance of vector multiplication instruction `VPMULUDQ`, there are two ports of Skylake CPU executing this instruction while only one port of Haswell CPU handling the same instruction. As a result, the throughput properties of `VPMULUDQ` are different on these two platforms, which is one instruction per cycle on Haswell and two instructions per cycle on Skylake.

Concerning AVX2 vector instructions, the operands are stored in 256-bit `YMM` vector registers, and each 256-bit operand can be regarded as an array of elements. A vector instruction deals with the elements of a `YMM` register through multiple parallel execution units. For example, `VPADDQ` takes four 64-bit wide parallel execution units and accomplishes four 64-bit additions at once. But AVX2 only supports up to 32-bit multipliers, for which `VPMULUDQ` computes the lower 32 bits of four 64-bit elements and yields four 64-bit products.

3 Vectorized Field Arithmetic

Field operations are fundamental components of the elliptic curve arithmetic. This section deals with (4×1)-way vectorized implementations of arithmetic in $\mathbb{F}_{2^{255}-19}$, and presents some relevant optimization techniques. Section 3.1 introduces the element vector set with radix-2^{29} representation and explain the reasons of this design. In Section 3.2, based on field element vector sets, we show how to perform (4×1)-way field operations efficiently.

3.1 Radix-2^{29} Element Vector Set

There have been multiple discussions about how to efficiently represent 255-bit field elements in the X25519 implementation [5, 7, 12], which state the choice of radix is always platform-dependent. Considering AVX2 extensions and our (4×1)-way strategy, we come up with a radix-2^{29} representation for the field element such that

$$f = f_0 + 2^{29} f_1 + 2^{58} f_2 + 2^{87} f_3 + 2^{116} f_4 + 2^{145} f_5 + 2^{174} f_6 + 2^{203} f_7 + 2^{232} f_8,$$

where each $0 \leq f_i < 2^{29}$. It is called radix-2^{29} representation since each *limb* f_i is 29 bits long, and it thus allows the field elements up to $9 \times 29 = 261$-bit

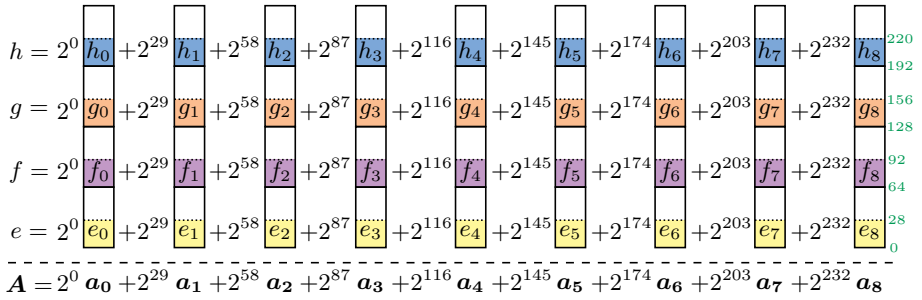

Fig. 1. Structure of a vector set $\boldsymbol{A}$ consisting of nine 256-bit limb vectors that contain four field elements altogether (i.e. four 29-bit limbs per vector).

during the computations. We then define a radix-2^{29} element vector set $\boldsymbol{A}$ as:

$$\begin{aligned}\boldsymbol{A} = [e, f, g, h] &= [\sum_{i=0}^{8} 2^{29i} e_i, \sum_{i=0}^{8} 2^{29i} f_i, \sum_{i=0}^{8} 2^{29i} g_i, \sum_{i=0}^{8} 2^{29i} h_i] \\ &= \sum_{i=0}^{8} 2^{29i} [e_i, f_i, g_i, h_i] = \sum_{i=0}^{8} 2^{29i} \boldsymbol{a_i} \quad \text{with} \quad \boldsymbol{a_i} = [e_i, f_i, g_i, h_i]. \end{aligned} \tag{1}$$

$\boldsymbol{A}$ is made up of nine 256-bit *limb vectors* $\boldsymbol{a_i}$, and it is consists of *four* field elements (e, f, g and h) altogether, which fits (4×1)-way field operations.

However, in most of the AVX2 implementations of X25519, e.g. [5] and [7], the radix-$2^{25.5}$ representation is recommended to represent a field element f such that

$$f = f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9,$$

where $0 \leq f_{2j} < 2^{26}$ and $0 \leq f_{2j+1} < 2^{25}$ for $0 \leq j \leq 4$, and totally ten limbs for an element. There are several reasons why we prefer radix-2^{29} to radix-$2^{25.5}$ in our implementation. The implementations of [5] and [7] perform the field operations in a (2×2)-way, which puts two limbs of each element in one 256-bit limb vector. In this setting, both radix-2^{29} and radix-$2^{25.5}$ make an element vector set possess five limb vectors. And radix-$2^{25.5}$ representation offers more available bits to delay the carry propagation. However, our implementation is in (4×1)-way. This means in an element vector set, there are nine limb vectors in radix-2^{29} but ten limb vectors in radix-$2^{25.5}$. The fewer limb vectors require fewer vector instructions and contribute to faster field operations. In addition, three available bits are sufficient for our software to delay the carry propagation and offer high performance.

Fig. 1 illustrates the structure of an element vector set $\boldsymbol{A}$, where elements are in four coloured 29-bit rows, and each column represents a limb vector $\boldsymbol{a_i}$. Our implementation executes four scalar multiplication instances in parallel so that integers e, f, g and h are from different instances, and there is no dependency

among four elements. The precise bit position of 256-bit limb vector is on the right side of the column of $\boldsymbol{a_8}$. Since `VPMULUDQ` instruction multiplies only lower 32 bits of each 64-bit lane of the input operands, we store each limb at the bit position from $64i$ to $64i+28$ for $0 \leq i \leq 3$. Besides, the radix-2^{29} representation provides three available bits to delay the carry propagation. During the computations of our software, the limb of elements can be more than 29-bit but always do not exceed 31-bit, i.e. at the bit position from $64i$ to $64i+30$ for $0 \leq i \leq 3$.

3.2 Implementing Field Operations with AVX2

We stipulate that all the operands here are radix-2^{29} element vector sets, and each limb of each field element is within 32-bit. Due to the radix-2^{29} representation, we set the modulus p in our field operations as $2^6 \cdot (2^{255} - 19)$ to facilitate computations. The simple C code of all the (4×1)-way vectorized field operations are shown in Listing 2 at Appendix A. In particular, the (4×1)-way field multiplication is described in Listing 1 in this section, and with a graphic illustration in Fig. 2.

Addition. The vectorized addition $\boldsymbol{R} = \boldsymbol{A} + \boldsymbol{B}$ is implemented in a straightforward way, where only nine `VPADDQ` instructions to compute $\boldsymbol{r_i} = \boldsymbol{a_i} + \boldsymbol{b_i}$ for $0 \leq i \leq 8$. In essence, we replace a modular addition by an ordinary addition without the modulo-p reduction. Our addition delays the carry propagation and allows each limb of sum $\boldsymbol{R}$ to expand one more bit.

Subtraction. We compute $r = 2p + a - b$ instead of $r = a - b$ for each instance to avoid getting any negative intermediate values. Besides, we developed two types of field subtraction: one is an ordinary subtraction, and the other one is a modular subtraction. We observed that our software does not strictly require each subtraction to perform the carry propagation and reduction operation. Therefore, if we use ordinary subtraction properly, i.e. do not cause overflow, the implementation performance will be improved. The ordinary subtraction is similar to the field addition, and it uses nine `VPADDQ` and `VPSUBQ` instructions. Compared to it, the modular subtraction is more costly due to the cost of carry propagation and reduction.

Multiplication. Modular multiplication is usually regarded as the most critical field operation due to a high frequency of use and relatively large latency (compared with other field operations), which deserves more care. Our design aims at minimising the sequential dependencies among the involved instructions. With this benefit, the processor can deal with instructions as much as in parallel (i.e. makes fully use of the different execution ports), whereby accelerates the entire modular multiplication. However, tuning code to weaken or even get rid of the encumbering from dependency chains sometimes will import extra instructions

Listing 1. Simple C implementation of (4×1)-way field multiplication

```
#include <immintrin.h>
#define ADD(X,Y) _mm256_add_epi64(X,Y)  /* VPADDQ */
#define MUL(X,Y) _mm256_mul_epu32(X,Y)  /* VPMULUDQ */
#define AND(X,Y) _mm256_and_si256(X,Y)  /* VPAND */
#define SRL(X,Y) _mm256_srli_epi64(X,Y) /* VPSRLQ */
#define BCAST(X) _mm256_set1_epi64x(X)  /* VPBROADCASTQ */
#define MASK29   0x1fffffff             /* mask of 29 LSBs */

void fp_mul(__m256i *r, const __m256i *a, const __m256i *b)
{
  int i, j, k; __m256i t[9], accu;

  /* 1st loop of the product-scanning multiplication */
  for (i = 0; i < 9; i++) {
    t[i] = BCAST(0);
    for(j = 0, k = i; k >= 0; j++, k--)
      t[i] = ADD(t[i], MUL(a[j], b[k]));
  }
  accu = SRL(t[8], 29);
  t[8] = AND(t[8], BCAST(MASK29));

  /* 2nd loop of the product-scanning multiplication */
  for (i = 9; i < 17; i++) {
    for (j = i-8, k = 8; j < 9; j++, k--)
      accu = ADD(accu, MUL(a[j], b[k]));
    r[i-9] = AND(accu, BCAST(MASK29));
    accu = SRL(accu, 29);
  }
  r[8] = accu;

  /* modulo reduction and conversion to 29-bit limbs */
  accu = BCAST(0);
  for (i = 0; i < 9; i++) {
    accu = ADD(accu, MUL(r[i], BCAST(64*19)));
    accu = ADD(accu, t[i]);
    r[i] = AND(accu, BCAST(MASK29));
    accu = SRL(accu, 29);
  }

  /* limbs in r[0] can finally be 30 bits long */
  r[0] = ADD(r[0], MUL(accu, BCAST(64*19)));
}
```

and in return slow down the software. Finding an optimal multiplication strategy, which reasonably schedules an instruction sequence and fully exploits the platform's parallel processing capability, is a challenging task.

Taking into account the different latency and throughput properties of various AVX2 instructions, we conducted experiments with a dozen self-developed variants of modular multiplication. All the variants use a *product-scanning* approach [11]. The distinctions among these variants include but are not limited to:

1. The modulo-q reduction is either *separated* or *interleaved* with the multiplication. If interleaved, how to interleave both.
2. Different plans of the carry propagation.
3. Whether and how to store intermediate values in local variables.

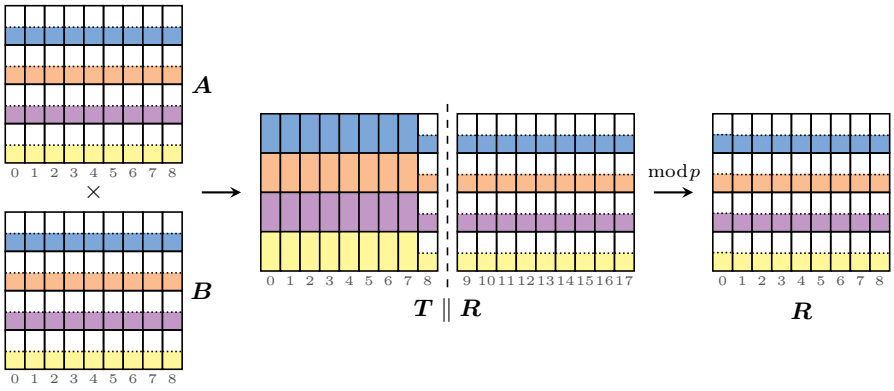

Fig. 2. (4×1)-way vectorized field multiplication.

At last, a benchmark of all the variants told us which one is the fastest. Listing 1 shows our (4×1)-way field multiplication, which performs the product-scanning multiplication and the modular reduction separately. It takes advantage of a local array `t`, to keep intermediate products generated in the first loop, whereby the sequential dependencies of intermediate products in that loop are thoroughly eliminated. But there are dependency chains in the second loop regarding updates of the accumulator `accu`. In Figure 2, a joint vector set $\boldsymbol{T} \parallel \boldsymbol{R}$ stores the intermediate product of $\boldsymbol{A} \times \boldsymbol{B}$, with corresponds to arrays `t` and `r` in Listing 1. After the product-scanning multiplication, a modulo-p reduction is performed and obtains the result $\boldsymbol{R}$.

We take the routine of a single lane as an example to analyze and prove there is no overflow. In our implementation, the *most extreme* case of field multiplication is that one operand a is an output of an ordinary subtraction and the other one b is a sum of the addition operation. The boundary of each limb of a and b are respectively $a_i < 2^{30.59}$ and $b_i < 2^{30.01}$. During the product-scanning multiplication, the theoretical maximal value appears in computing

$$t_7 = a_0b_7 + a_1b_6 + a_2b_5 + a_3b_4 + a_4b_3 + a_5b_2 + a_6b_1 + a_7b_0,$$

and therefore $t_7 < 2^{63.61}$. Yet, the theoretical maximal value of the whole modular multiplication appears in the reduction operation, i.e. computing $accu' = accu + r_7 \times 19 \times 64 + t_7$ (corresponding to lines 34 and 35 in Listing 1). Obviously, $accu < 2^{35}$ and $r_7 < 2^{29}$ so that $accu' < 2^{63.62}$, which proves that no intermediate value exceeds 64 bits during the field multiplication.

Squaring. Modular squaring $\boldsymbol{R} = \boldsymbol{A}^2 \bmod p$ can be considered as the special case of modular multiplication where two operands are the same. In general, squaring is faster than multiplication, because the product of $\boldsymbol{a_i a_j}$ where $i \neq j$ is computed twice during the squaring. We can thereby compute products $\boldsymbol{a_i a_j}$ and double them by the bit-wise shift to save numerous instructions. In the

Op.	ParLadStep	Instance 0	Instance 1	Instance 2	Instance 3
1	$\boldsymbol{T} \leftarrow \boldsymbol{X_P} + \boldsymbol{Z_P}$	$e \leftarrow x_{\mathcal{A}} + z_{\mathcal{A}}$	$f \leftarrow x_{\mathcal{B}} + z_{\mathcal{B}}$	$g \leftarrow x_{\mathcal{C}} + z_{\mathcal{C}}$	$h \leftarrow x_{\mathcal{D}} + z_{\mathcal{D}}$
2	$\boldsymbol{X_P} \leftarrow \boldsymbol{X_P} - \boldsymbol{Z_P}$	$x_{\mathcal{A}} \leftarrow x_{\mathcal{A}} - z_{\mathcal{A}}$	$x_{\mathcal{B}} \leftarrow x_{\mathcal{B}} - z_{\mathcal{B}}$	$x_{\mathcal{C}} \leftarrow x_{\mathcal{C}} - z_{\mathcal{C}}$	$x_{\mathcal{D}} \leftarrow x_{\mathcal{D}} - z_{\mathcal{D}}$
3	$\boldsymbol{T'} \leftarrow \boldsymbol{X_Q} + \boldsymbol{Z_Q}$	$e' \leftarrow x_{\mathcal{I}} + z_{\mathcal{I}}$	$f' \leftarrow x_{\mathcal{J}} + z_{\mathcal{J}}$	$g' \leftarrow x_{\mathcal{K}} + z_{\mathcal{K}}$	$h' \leftarrow x_{\mathcal{L}} + z_{\mathcal{L}}$
4	$\boldsymbol{X_Q} \leftarrow \boldsymbol{X_Q} - \boldsymbol{Z_Q}$	$x_{\mathcal{I}} \leftarrow x_{\mathcal{I}} - z_{\mathcal{I}}$	$x_{\mathcal{J}} \leftarrow x_{\mathcal{J}} - z_{\mathcal{J}}$	$x_{\mathcal{K}} \leftarrow x_{\mathcal{K}} - z_{\mathcal{K}}$	$x_{\mathcal{L}} \leftarrow x_{\mathcal{L}} - z_{\mathcal{L}}$
5	$\boldsymbol{Z_P} \leftarrow \boldsymbol{T}^2$	$x_{\mathcal{A}} \leftarrow e^2$	$x_{\mathcal{B}} \leftarrow f^2$	$x_{\mathcal{C}} \leftarrow g^2$	$x_{\mathcal{D}} \leftarrow h^2$
6	$\boldsymbol{Z_Q} \leftarrow \boldsymbol{T'} \times \boldsymbol{X_P}$	$z_I \leftarrow e' \times x_{\mathcal{A}}$	$z_{\mathcal{J}} \leftarrow f' \times x_{\mathcal{B}}$	$z_{\mathcal{K}} \leftarrow g' \times x_{\mathcal{C}}$	$z_L \leftarrow h' \times x_{\mathcal{D}}$
7	$\boldsymbol{T'} \leftarrow \boldsymbol{X_Q} \times \boldsymbol{T}$	$e' \leftarrow x_{\mathcal{I}} \times e$	$f' \leftarrow x_{\mathcal{J}} \times f$	$g' \leftarrow x_{\mathcal{K}} \times g$	$h' \leftarrow x_{\mathcal{L}} \times h$
8	$\boldsymbol{T} \leftarrow \boldsymbol{X_P}^2$	$e \leftarrow x_{\mathcal{A}}{}^2$	$f \leftarrow x_{\mathcal{B}}{}^2$	$g \leftarrow x_{\mathcal{C}}{}^2$	$h \leftarrow x_{\mathcal{D}}{}^2$
9	$\boldsymbol{X_P} \leftarrow \boldsymbol{Z_P} \times \boldsymbol{T}$	$x_{\mathcal{A}} \leftarrow z_{\mathcal{A}} \times e$	$x_{\mathcal{B}} \leftarrow z_{\mathcal{B}} \times f$	$x_{\mathcal{C}} \leftarrow z_{\mathcal{C}} \times g$	$x_{\mathcal{D}} \leftarrow z_{\mathcal{D}} \times h$
10	$\boldsymbol{T} \leftarrow \boldsymbol{Z_P} - \boldsymbol{T}$	$e \leftarrow z_{\mathcal{A}} - e$	$f \leftarrow z_{\mathcal{B}} - f$	$g \leftarrow z_{\mathcal{C}} - g$	$h \leftarrow z_{\mathcal{D}} - h$
11	$\boldsymbol{X_Q} \leftarrow \boldsymbol{T} \times \boldsymbol{\pi}$	$x_{\mathcal{I}} \leftarrow e \times \pi$	$x_{\mathcal{J}} \leftarrow f \times \pi$	$x_{\mathcal{K}} \leftarrow g \times \pi$	$x_{\mathcal{L}} \leftarrow h \times \pi$
12	$\boldsymbol{X_Q} \leftarrow \boldsymbol{X_Q} + \boldsymbol{Z_P}$	$x_{\mathcal{I}} \leftarrow x_{\mathcal{I}} + z_{\mathcal{A}}$	$x_{\mathcal{J}} \leftarrow x_{\mathcal{J}} + z_{\mathcal{B}}$	$x_{\mathcal{K}} \leftarrow x_{\mathcal{K}} + z_{\mathcal{C}}$	$x_{\mathcal{L}} \leftarrow x_{\mathcal{L}} + z_{\mathcal{D}}$
13	$\boldsymbol{Z_P} \leftarrow \boldsymbol{X_Q} \times \boldsymbol{T}$	$z_{\mathcal{A}} \leftarrow x_{\mathcal{I}} \times e$	$z_{\mathcal{B}} \leftarrow x_{\mathcal{J}} \times f$	$z_{\mathcal{C}} \leftarrow x_{\mathcal{K}} \times g$	$z_{\mathcal{D}} \leftarrow x_{\mathcal{L}} \times h$
14	$\boldsymbol{T} \leftarrow \boldsymbol{T'} + \boldsymbol{Z_Q}$	$e \leftarrow e' + z_{\mathcal{I}}$	$f \leftarrow f' + z_{\mathcal{J}}$	$g \leftarrow g' + z_{\mathcal{K}}$	$h \leftarrow h' + z_{\mathcal{L}}$
15	$\boldsymbol{X_Q} \leftarrow \boldsymbol{T}^2$	$x_{\mathcal{I}} \leftarrow e^2$	$x_{\mathcal{J}} \leftarrow f^2$	$x_{\mathcal{K}} \leftarrow g^2$	$x_{\mathcal{L}} \leftarrow h^2$
16	$\boldsymbol{T} \leftarrow \boldsymbol{T'} - \boldsymbol{Z_Q}$	$e \leftarrow e' - z_{\mathcal{I}}$	$f \leftarrow f' - z_{\mathcal{J}}$	$g \leftarrow g' - z_{\mathcal{K}}$	$h \leftarrow h' - z_{\mathcal{L}}$
17	$\boldsymbol{T'} \leftarrow \boldsymbol{T}^2$	$e' \leftarrow e^2$	$f' \leftarrow f^2$	$g' \leftarrow g^2$	$h' \leftarrow h^2$
18	$\boldsymbol{Z_Q} \leftarrow \boldsymbol{T'} \times \boldsymbol{\Omega}$	$z_I \leftarrow e' \times \alpha$	$z_{\mathcal{J}} \leftarrow f' \times \beta$	$z_{\mathcal{K}} \leftarrow g' \times \gamma$	$z_{\mathcal{L}} \leftarrow h' \times \delta$

Fig. 3. Parallel Montgomery ladder step of our implementation

beginning, we also developed several variants of modular squaring according to the same rationales of designing modular multiplications. Likewise, the modular squaring based on the same modular multiplication algorithm that we described before, i.e. Listing 1, possesses the best performance.

4 The (4×1)-Way Scalar Multiplication

The complete ECDH relies on two types of point scalar multiplications that one operates on a fixed base point (for key generation) and the other one works with variable base points (for computing the shared secret). As stated before, both fixed-base and variable-base scalar multiplications of our software are in (4×1)-parallel way.

4.1 Variable-Base Scalar Multiplication

Montgomery ladder [14] is the most commonly used algorithm for variable-base scalar multiplication, which performs one *ladder step* for each scalar bit. In essence, a ladder step consists of a differential point addition and a point doubling, only concerning the x and z coordinates of points in projective coordinates. Moreover, the Montgomery ladder possesses constant running time since each ladder step is performed by a fixed instruction sequence.

Point Vector Set. Similar to the element vector set, the point vector set is also 4-way i.e. each point vector set contains four points altogether. For example, a

point vector set in affine coordinates is:

$$\begin{aligned}\boldsymbol{\mathcal{P}} &= [\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}] = [(x_{\mathcal{A}}, y_{\mathcal{A}}), (x_{\mathcal{B}}, y_{\mathcal{B}}), (x_{\mathcal{C}}, y_{\mathcal{C}}), (x_{\mathcal{D}}, y_{\mathcal{D}})] \\ &= ([x_{\mathcal{A}}, x_{\mathcal{B}}, x_{\mathcal{C}}, x_{\mathcal{D}}], [y_{\mathcal{A}}, y_{\mathcal{B}}, y_{\mathcal{C}}, y_{\mathcal{D}}]) = (\boldsymbol{X}_{\boldsymbol{\mathcal{P}}}, \boldsymbol{Y}_{\boldsymbol{\mathcal{P}}}). \end{aligned} \tag{2}$$

Similarly, the point vector set in projective coordinates is $\boldsymbol{\mathcal{P}} = [\boldsymbol{X}_{\boldsymbol{\mathcal{P}}}, \boldsymbol{Y}_{\boldsymbol{\mathcal{P}}}, \boldsymbol{Z}_{\boldsymbol{\mathcal{P}}}]$.

(4×1)-Way Montgomery Ladder. A conventional Montgomery ladder updates two points $\mathcal{P}$ and $\mathcal{Q}$, or rather their x- and z-coordinates, by using the base point's x-coordinate ω (i.e. public key of the other party) and the bit-string of the scalar. As for the ladder step, it updates the x- and z-coordinates of two points, i.e. $(x_{\mathcal{P}}, z_{\mathcal{P}})$ and $(x_{\mathcal{Q}}, z_{\mathcal{Q}})$, with also the input of ω. However, in our implementation, they should be switched to the corresponding vector sets. In detail, we define the conventional ladder step as $(\mathcal{P}, \mathcal{Q}) \leftarrow \textsc{LadStep}(\mathcal{P}, \mathcal{Q}, \omega)$ while the parallel Montgomery ladder step as $(\boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{Q}}) \leftarrow \textsc{ParLadStep}(\boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{Q}}, \boldsymbol{\Omega})$. Suppose that two point vector sets $\boldsymbol{\mathcal{P}} = [\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}]$ and $\boldsymbol{\mathcal{Q}} = [\mathcal{I}, \mathcal{J}, \mathcal{K}, \mathcal{L}]$, and element vector set $\boldsymbol{\Omega} = [\alpha, \beta, \gamma, \delta]$, the relation between LadStep and ParLadStep can be explained as follows:

$$\begin{aligned} &\textsc{ParLadStep}(\boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{Q}}, \boldsymbol{\Omega}) \\ =&\textsc{ParLadStep}([\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}], [\mathcal{I}, \mathcal{J}, \mathcal{K}, \mathcal{L}], [\alpha, \beta, \gamma, \delta]) \\ =&[\textsc{LadStep}(\mathcal{A}, \mathcal{I}, \alpha), \textsc{LadStep}(\mathcal{B}, \mathcal{J}, \beta), \textsc{LadStep}(\mathcal{C}, \mathcal{K}, \gamma), \textsc{LadStep}(\mathcal{D}, \mathcal{L}, \delta)] \\ \rightarrow&[(\mathcal{A}, \mathcal{I}), (\mathcal{B}, \mathcal{J}), (\mathcal{C}, \mathcal{K}), (\mathcal{D}, \mathcal{L})] = ([\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}], [\mathcal{I}, \mathcal{J}, \mathcal{K}, \mathcal{L}]) = (\boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{Q}}) \end{aligned} \tag{3}$$

The details of Equation (3) and the instruction sequence of a ladder step are shown in Fig. 3, where the ordinary subtraction $r = 2p + a - b$ is highlighted (the 2nd and 10th operation). There are also two temporary element vector sets $\boldsymbol{T} = [e, f, g, h]$ and $\boldsymbol{T'} = [e', f', g', h']$. Furthermore, a vector $\boldsymbol{\pi} = [\pi, \pi, \pi, \pi]$ contains four same constants that are the public parameters of curve, i.e. each $\pi = (a - 2)/4 = 121665$ for Curve25519.

4.2 Fixed-Base Scalar Multiplication

The fixed-base scalar multiplication $\mathcal{R} = k\mathcal{B}$ is generally carried out on Ed25519, a twisted Edwards curve that is birationally equivalent to Curve25519, to gain a speed improvement. This scalar multiplication takes a fixed base point $\mathcal{B}$ with the y-coordinate of $4/5$ (corresponding to the standard base point in Montgomery curve), and the scalar k is a random 255-bit integer. In detail, the scalar k can be written as $\Sigma_{i=0}^{63} 16^i k_i$ where $k_i \in \{-8, -7, \ldots, 7\}$. $\mathcal{R} = k\mathcal{B}$ is therefore computed through

$$\mathcal{R} = \sum_{i=0}^{63} k_i \cdot 16^i \mathcal{B}. \tag{4}$$

An efficient and popular technique regarding the computation of Equation (4) has been researched by Bernstein et al. in [3], which takes advantage of a precomputed look-up table. The table stores eight multiples of each $16^i\mathcal{B}$, i.e.

$\{16^{2i}\mathcal{B}, 2 \cdot 16^{2i}\mathcal{B}, \ldots, 8 \cdot 16^{2i}\mathcal{B}\}$, and it thus consists of totally $64 \times 8 = 512$ points. One can rapidly obtain each $|k_i| \cdot 16^i\mathcal{B}$ by searching $|k_i|$ with i in the table with constant time, and then compute $k_i \cdot 16^i\mathcal{B}$ according to the sign of k_i. This table-based method saves considerable computations of point arithmetic and speeds up the software. However, the look-up table accordingly consumes some memory. A better trade-off has also been proposed in [3], which divides the Equation (4) into two parts:

$$\mathcal{R} = \sum_{i=0}^{31} k_{2i} \cdot 16^{2i}\mathcal{B} + 16 \cdot \sum_{i=0}^{31} k_{2i+1} \cdot 16^{2i}\mathcal{B}. \tag{5}$$

At the cost of four point doublings, it halves the size of the table (from 64×8 to 32×8) and only brings little speed loss. Our implementation is developed according to this method of Equation (5) and perform the parallel fixed-base scalar multiplication in (4×1)-way, i.e.

$$\boldsymbol{\mathcal{R}} = \sum_{i=0}^{31} \boldsymbol{k_{2i}} \cdot 16^{2i}\boldsymbol{\mathcal{B}} + 16 \cdot \sum_{i=0}^{31} \boldsymbol{k_{2i+1}} \cdot 16^{2i}\boldsymbol{\mathcal{B}}, \tag{6}$$

where $\boldsymbol{\mathcal{B}} = [\mathcal{B}, \mathcal{B}, \mathcal{B}, \mathcal{B}]$; i.e., every instance uses the standard base point. Hence, the table in our software need not to be constructed as a vectorized table i.e. made up of four duplicate tables. We can just create a conventional precomputed table and use the `VPBROADCASTQ` instruction to broadcast values to four instances.

Look-Up Table. As explained before, the precomputed table stores $32 \times 8 = 256$ points, all of which are the multiples of base $\mathcal{B}$. We take advantage of *full-radix* representation [7] instead of the reduced-radix to store each point in the table, i.e. the limb of point's coordinate is 32 bits long. By this way, each coordinate (i.e. a field element) contains eight 32-bit limbs. Therefore, each point takes 96 bytes while the whole look-up table amounts to 24 kB. Moreover, the points in the table are in extended affine coordinates (u, v, w):

$$u = (x + y)/2, \quad v = (y - x)/2, \quad w = dxy,$$

where d is the public parameter of curve. This representation of coordinate facilitates a unified mixed point addition [13] which is illustrated to be very efficient on twisted Edwards curve. As a result, a point of the table is stored as

$$\mathcal{P} = (u_\mathcal{P}, v_\mathcal{P}, w_\mathcal{P}) = \begin{pmatrix} \{\underline{u_{\mathcal{P}_0}, u_{\mathcal{P}_1}}, \ \underline{u_{\mathcal{P}_2}, u_{\mathcal{P}_3}}, \ \underline{u_{\mathcal{P}_4}, u_{\mathcal{P}_5}}, \ \underline{u_{\mathcal{P}_6}, u_{\mathcal{P}_7}}\}, \\ \{\underline{v_{\mathcal{P}_0}, v_{\mathcal{P}_1}}, \ \underline{v_{\mathcal{P}_2}, v_{\mathcal{P}_3}}, \ \underline{v_{\mathcal{P}_4}, v_{\mathcal{P}_5}}, \ \underline{v_{\mathcal{P}_6}, v_{\mathcal{P}_7}}\}, \\ \{\underline{w_{\mathcal{P}_0}, w_{\mathcal{P}_1}}, \ \underline{w_{\mathcal{P}_2}, w_{\mathcal{P}_3}}, \ \underline{w_{\mathcal{P}_4}, w_{\mathcal{P}_5}}, \ \underline{w_{\mathcal{P}_6}, w_{\mathcal{P}_7}}\} \end{pmatrix}. \tag{7}$$

Since we assign the 64-bit lane of a 256-bit `YMM` register to each instance, when using `VPBROADCASTQ` instructions to broadcast the values of the table to `YMM` registers, each instruction deals with two 32-bit words that appear over the same underline. This allows our implementation to perform less subsequent instructions and reduces latency.

Table 1. CPU-cycle counts of (4×1)-way field and point operations.

Domain	Operation	Faz-H. et al. [7]		**This Work**	
		Haswell	Skylake	Haswell	Skylake
$\mathbb{F}_p$	Addition	12	12	11	11
	Ord. Subtraction	n/a	n/a	14	12
	Mod. Subtraction	n/a	n/a	32	31
	Multiplication	159	105	122	88
	Squaring	114	85	87	65
twisted Edwards curve	Point Addition	1096	833	965	705
	Point Doubling	n/a	n/a	830	624
	Table Query	208	201	218	205
Montgomery curve	Ladder Step	1660†	1372†	1118	818

† The ladder step of [7] only works for a single instance. For an intuitive comparison, we quadruple the latency of ladder step reported in [7].

(4×1)-Way Point Operations. There are three types of point operations in a fixed-base scalar multiplication, i.e. point addition, point doubling and table query (i.e. a table-based point multiplication). All of them follow the same (4×1)-parallel way of how a ladder step behaved, where four instances are inherently independent with each other. Analogously, each type of operations has a fixed instruction sequence and works on point vector sets. We also make use of two types of field subtraction to speed up the point addition and doubling. The simple C implementations of point addition and point doubling are shown in Listing 3 at Appendix B.

5 Performance Evaluation and Comparison

Our source codes have the fixed instructions sequence without any conditional branches (i.e. *if-else* statement), so the implementation is constant-time and secure against timing attacks. We measured the performance of our software on the following two processors:

- a *Haswell* Intel Core i7-4710HQ CPU clocked at 2.5 GHz;
- a *Skylake* Intel Core i5-6360U CPU clocked at 2.0 GHz.

We used Clang compiler with a version 10.0.0 to compile our source codes on both processors. The turbo-boost and hyper-threading features were disabled during performance measurements.

Table 1 lists the latency of field and point arithmetic operations of our software and gives a comparison to (4×1)-way operations developed by Faz-Hernández et al. that reported in [7]. For the most performance-critical field operation of X25519, i.e. modular multiplication, our implementation outperforms that of [7], which respectively costs 37 and 17 clock cycles fewer on Haswell and Skylake microarchitectures. It is not surprising that our faster field operations contributed to faster point arithmetic at a higher level. For example, a point addition on twisted Edwards curve of our software requires 965 clock cycles on

Table 2. The performance of our software on Haswell i7-4710HQ CPU and Skylake i5-6360U CPU.

Platform	CPU Frequency	Key Generation		Shared Secret		Table Size
		Latency	Throughput	Latency	Throughput	
Haswell	2.5 GHz	104,579 cycles	95,568 ops/sec	329,455 cycles	30,336 ops/sec	24 kB
Skylake	2.0 GHz	80,249 cycles	99,363 ops/sec	246,636 cycles	32,318 ops/sec	24 kB

Table 3. The performance comparison of X25519 AVX2 implementations on Haswell CPU. The throughput data were measured at a unified frequency of 2.5 GHz.

Work	Impl.	CPU	Compiler	Key Generation		Shared Secret	
				Latency [cycles]	Throughput [ops/sec]	Latency [cycles]	Throughput [ops/sec]
Faz-H. et al. [7]	(2×2)-way	i7-4770	Clang 5.0.2	43,700	57,208†	121,000	20,661†
	(2×2)-way	i7-4710HQ	Clang 10.0.0	41,938	59,575	121,499	20,563
Nath et al. [15]	(4×1)-way	i7-6500U	GCC 7.3.0	100,127	24,968†	120,108	20,815†
	(4×1)-way	i7-4710HQ	GCC 8.4.0	100,669	24,820	120,847	20,676
This work	(4×1)-way	i7-4710HQ	Clang 10.0.0	104,579⋆	**95,568** 60.4%	329,455⋆	**30,336** 45.7%

† [7] and [15] do not provide the throughput data. We list their theoretical throughput that obtained through the frequency of 2.5 GHz divided by the latency.
⋆ The latency data of our implementation is the latency of executing four instances.

Haswell CPU, which is faster than it of [7] by a factor of 14%. Our (4×1)-way Montgomery ladder step takes less than one thousand clock cycles on Skylake CPU, which is only 818 cycles.

Overall performance of our implementation on Haswell and Skylake platforms are shown in Table 2. Since our software performs four scalar multiplications in parallel, the latency here includes four computations of key generation or shared secret. Thanks to a 24 kB precomputed look-up table, the key generation is over three times faster than computing the shared secret on both platforms. As for throughput, our software can compute more than 95 thousands key generations or 30 thousands shared secrets per second on a Haswell CPU clocked at 2.5 GHz. When evaluated at the same CPU frequency, the throughput capacity of our software is around 30% stronger on Skylake than on Haswell platform.

Table 3 compares our work with recent X25519 implementations with AVX2 on the Haswell microarchitecture. In order to avoid as much as possible the influence of different compilers and different processors, we downloaded and compiled the source codes of [7] and [15] in our own experiment environment and measured their corresponding performance data. Both performance data reported in their papers and measured by ourselves are presented in the table, and they are listed respectively in upper and lower neighbouring rows. The performance data measured by ourselves can also be easily recognised according to the CPU type and a compiler of more recent version. Notably, we tried with Clang 10.0.0 to compile Nath et al.'s source codes, but the performance is not efficient as

Table 4. The performance comparison of X25519 AVX2 implementations on Skylake CPU. The throughput data were measured at a unified frequency of 2.0 GHz.

Work	Impl.	CPU	Compiler	Key Generation		Shared Secret	
				Latency [cycles]	Throughput [ops/sec]	Latency [cycles]	Throughput [ops/sec]
Faz-H. et al. [7]	(2×2)-way	i7-6700K	Clang 5.0.2	34,500	57,971‡	99,400	20,150‡
	(2×2)-way	i5-6360U	Clang 10.0.0	35,629	55,955	95,129	20,939
Hisil et al. [12]	(4×1)-way	i9-7900X	GCC 5.4	n/a	n/a	98,484	20,308†
	(4×1)-way	i5-6360U	GCC 8.4.0	n/a	n/a	116,595	16,656
Nath et al. [15]	(4×1)-way	i7-6500U	GCC 7.3.0	84,047	23,796†	95,437	20,956†
	(4×1)-way	i5-6360U	GCC 8.4.0	82,054	24,406	93,657	21,168
This work	(4×1)-way	i5-6360U	Clang 10.0.0	80,249⋆	**99,363** 71.4%	246,636⋆	**32,318** 52.7%

† [12] and [15] do not provide the throughput data. We list their theoretical throughput obtained through the frequency of 2.0 GHz divided by the latency.
‡ [7] provides the throughput data measured on a CPU clocked at 3.6 GHz. We list the theoretical throughput obtained through their original result scaled with a factor of 5/9 for intuitive comparison.
⋆ The latency data of our implementation is the latency of executing four instances.

expected. A possible reason is that Nath et al. "tuned" their code to be fast with GCC, and the software will have significant performance changes under the compilation of two different compilers. Therefore, we used the GCC 8.4.0 that was released in March 2020 to compile their source codes and measured the performance. Since the throughput is also CPU-frequency-dependent, we set a unified frequency of 2.5 GHz for the comparison, which helps us only compare the throughput of software themselves. Besides, it is able to calculate a theoretical throughput that divides the CPU frequency by the latency of a single instance. The implementation of [7] performs (2×2)-way field operations but also takes advantage of a (4×1)-way table query so that its key generation is also very efficient. Nevertheless, in terms of key generation, the throughput capacity of our software outperforms [7] by a factor of 60.4%. As for shared secret computations, our implementation reaches a 45.7% throughput improvement compared to the work of Nath et al. [15] which are implemented in assembly language.

The comparison between our implementation and others on Skylake platform is reported in Table 4. Analogously, we present both the performance data in original literatures of [7, 12, 15] and measured by ourselves. Our implementation generates around 100 thousands key pairs per second on a Skylake CPU clocked at 2.0 GHz, which significantly outperforms [7] by 71.4% for the throughput capability. The currently best latency-optimized variable-base scalar multiplication was proposed by Nath et al. in [15]. And the throughput of this implementation is a little bit higher when measured with a higher version of GCC by ourselves. Nevertheless, the throughput of our implementation is still significantly higher than it by a factor of 45.7%.

Moreover, both Table 3 and 4 indicate that a higher version of the compiler cannot be determined to bring a positive or negative effect for the implementation. For example in Table 4, Hisil et al.'s implementation became slower when

using GCC 8.4.0 instead of GCC 5.4 to compile their source codes. Yet most of the other implementations saw improvements when using a higher version compiler.

6 Conclusion

SIMD instructions are trending to support larger operand and larger register size. Notably, RISC-V architecture prepares to support up to 16384-bit vectors. It is an urgent task to explore how can we fully exploit such massive parallel processing capabilities in the ECC. When looking at Montgomery or twisted Edwards curves and optimizing for low latency with SIMD instructions, it is usually limited to execute four field multiplications in parallel, and more does not seem to be possible due to sequential dependencies. In this paper, we propose a new direction on how to take advantage of this computing resource in ECC schemes. Our work optimizes the cryptographic software possessing the fixed instruction sequence to perform multiple instances in parallel, which significantly improves the throughput of the scheme on processors. Because four instances in our implementation are independent with each other, our approach removes the sequential dependencies among elements of a vector register. It is therefore easily extensible to processors with more advanced vector instruction sets, and we can exploit the full parallelism of current and future SIMD instructions. For example, our software can be slightly modified to run eight instances on a processor with AVX512 extension to obtain the higher throughput. On the other hand, our software showed high performance on different platforms. On both the Haswell and Skylake microarchitectures, it immensely improves the throughput of ECC schemes. In a practical network environment, the server-side of communication usually need to deal with a large number of computations of cryptographic primitives per second. The throughput capacity of a scheme somewhat determines the quality of network communication. Our method and implementation provide inspiring help for the server-side to utilize elliptic curve cryptographic schemes more efficiently.

Acknowledgements. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM).

References

1. D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography — PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer Verlag, 2006.
2. D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. Twisted Edwards curves. In S. Vaudenay, editor, *Progress in Cryptology — AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer Verlag, 2008.

3. D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Sept. 2012.
4. 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.
5. T. Chou. Sandy2x: New curve25519 speed records. In O. Dunkelman and L. Keliher, editors, *Selected Areas in Cryptography – SAC 2015*, pages 145–160, Cham, 2016. Springer International Publishing.
6. A. Faz-Hernández and J. López. Fast implementation of Curve25519 using AVX2. In K. E. Lauter and F. Rodríguez-Henríquez, editors, *Progress in Cryptology — LATINCRYPT 2015*, volume 9230 of *Lecture Notes in Computer Science*, pages 329–345. Springer Verlag, 2015.
7. A. Faz-Hernández, J. López, and R. Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Trans. Math. Softw.*, 45(3), July 2019.
8. P. Grabher, J. Großschädl, and D. Page. On software parallel implementation of cryptographic pairings. In R. M. Avanzi, L. Keliher, and F. Sica, editors, *Selected Areas in Cryptography — SAC 2008*, volume 5381 of *Lecture Notes in Computer Science*, pages 35–50. Springer Verlag, 2009.
9. S. Gueron and V. Krasnov. Software implementation of modular exponentiation, using advanced vector instructions architectures. In F. Özbudak and F. Rodríguez-Henríquez, editors, *Arithmetic of Finite Fields — WAIFI 2012*, volume 7369 of *Lecture Notes in Computer Science*, pages 119–135. Springer Verlag, 2012.
10. T. R. Halfhill. RISC-V vectors know no limits. Linley Newsletter, available online at `http://www.linleygroup.com/newsletters/newsletter_detail.php?num=6154`, 2020.
11. D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.
12. H. Hisil, B. Egrice, and M. Yassi. Fast 4 way vectorized ladder for the complete set of montgomery curves. Cryptology ePrint Archive, Report 2020/388, 2020. `https://eprint.iacr.org/2020/388`.
13. H. Hişil, K. K.-H. Wong, G. Carter, and E. Dawson. Twisted Edwards curves revisited. In J. Pieprzyk, editor, *Advances in Cryptology — ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 326–343. Springer Verlag, 2008.
14. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, Jan. 1987.
15. K. Nath and P. Sarkar. Efficient 4-way vectorizations of the montgomery ladder. Cryptology ePrint Archive, Report 2020/378, 2020. `https://eprint.iacr.org/2020/378`.

A Implementation of Vectorized Filed Operations

Listing 2. Simple C implementation of (4×1)-way vectorized field operations

```
#include <immintrin.h>
#define ADD(X,Y) _mm256_add_epi64(X,Y)  /* VPADDQ */
#define SUB(X,Y) _mm256_sub_epi64(X,Y)  /* VPSUBQ */
#define MUL(X,Y) _mm256_mul_epu32(X,Y)  /* VPMULUDQ */
#define AND(X,Y) _mm256_and_si256(X,Y)  /* VPAND */
#define SRL(X,Y) _mm256_srli_epi64(X,Y) /* VPSRLQ */
#define SLL(X,Y) _mm256_slli_epi64(X,Y) /* VPSLLQ */
#define BCAST(X) _mm256_set1_epi64x(X)  /* VPBROADCASTQ */
#define MASK29   0x1fffffff             /* mask of 29 LSBs */

/* field addition */
void fp_add(__m256i *r, const __m256i *a, const __m256i *b)
{
  for (int i = 0; i < 9; i++) r[i] = ADD(a[i], b[i]);
}

/* field subtraction (without a carry propagation) */
void fp_sub(__m256i *r, const __m256i *a, const __m256i *b)
{
  /* subtraction loop */
  r[0] = ADD(BCAST(2*0x1ffffb40), SUB(a[0], b[0]);
  for (int i = 1; i < 9; i++)
    r[i] = ADD(BCAST(2*0x1fffffff), SUB(a[i], b[i]);
}

/* field subtraction (with a carry propagation) */
void fp_sbc(__m256i *r, const __m256i *a, const __m256i *b)
{
  /* subtraction loop */
  r[0] = ADD(BCAST(2*0x1ffffb40), SUB(a[0], b[0]);
  for (int i = 1; i < 9; i++)
    r[i] = ADD(BCAST(2*0x1fffffff), SUB(a[i], b[i]);

  /* carry propagation and conversion to 29-bit limbs*/
  for (int i = 1; i < 9; i++) {
    r[i] = ADD(r[i], SRL(r[i-1], 29));
    r[i-1] = AND(r[i-1], BCAST(MASK29));
  }

  /* limbs in r[0] can finally be 30 bits long */
  r[0] = ADD(r[0], MUL(BCAST(64*19), SRL(r[8], 29)));
  r[8] = AND(r[8], BCAST(MASK29));
}

/* field squaring */
void fp_sqr(__m256i *r, const __m256i *a)
{
  int i, j, k; __m256i t[9], accu, temp;

  /* 1st loop of the product-scanning squaring */
  t[0] = MUL(a[0],a[0]);
  for (i = 1; i < 9; i++) {
    t[i] = BCAST(0);
    for (j = 0, k = i; j < k; j++, k--)
      t[i] = ADD(t[i], MUL(a[j], a[k]));
    t[i] = SLL(t[i], 1);
    if (!(i&1)) t[i] = ADD(t[i], MUL(a[j], a[j]));
  }
  accu = SRL(t[8], 29);
  t[8] = AND(t[8], BCAST(MASK29));

  /* 2nd loop of the product-scanning squaring */
```

```
  for (i = 9; i < 16; i++) {
    temp = BCAST(0);
    for (j = i-8, k = 8; j < k; j++, k--)
      temp = ADD(r[i-9], MUL(a[j], a[k]));
    accu = ADD(accu, SLL(temp, 1));
    if (!(i&1)) accu = ADD(accu, MUL(a[j], a[j]));
    r[i-9] = AND(accu, BCAST(MASK29));
    accu = SRL(accu, 29);
  }
  accu = ADD(accu, MUL(a[8], a[8]));
  r[7] = AND(accu, BCAST(MASK29));
  r[8] = SRL(accu, 29);

  /* modulo reduction and conversion to 29-bit limbs */
  accu = BCAST(0);
  for (i = 0; i < 9; i++){
    accu = ADD(accu, MUL(r[i], BCAST(64*19))));
    accu = ADD(accu, t[i]);
    r[i] = AND(accu, BCAST(MASK29));
    accu = SRL(accu, 29);
  }

  /* limbs in r[0] can finally be 30 bits long */
  r[0] = ADD(r[0], MUL(accu, BCAST(64*19)));
}
```

B Implementation of (4×1)-Way Point Operations

Listing 3. Simple C implementation of (4×1)-way point operations

```
/**
 * @brief Point addition.
 *
 * @details
 * Unified mixed addition R = P + Q on a twisted Edwards
 * curve with a = -1.
 *
 * @param R Point in extended projective coordinates
 *          [x, y, z, e, h], e*h = t = x*y/z
 * @param P Point in extended projective coordinates
 *          [x, y, z, e, h], e*h = t = x*y/z
 * @param Q Point in extended affine coordinates
 *          [(y+x)/2, (y-x)/2, d*x*y]
 */
void point_add(ExtPoint *R, ExtPoint *P, ProPoint *Q)
{
  __m256i t[9];

  fp_mul(t, P->e, P->h);            /* T = E_P × H_P   */
  fp_sub(R->e, P->y, P->x);         /* E_R = Y_P − X_P */
  fp_add(R->h, P->y, P->x);         /* H_R = Y_P + X_P */
  fp_mul(R->x, R->e, Q->y);         /* X_R = E_R × Y_Q */
  fp_mul(R->y, R->h, Q->x);         /* Y_R = H_R × X_Q */
  fp_sub(R->e, R->y, R->x);         /* E_R = Y_R − X_R */
  fp_add(R->h, R->y, R->x);         /* H_R = Y_R + X_R */
  fp_mul(R->x, t, Q->z);            /* X_R = T × Z_Q   */
  fp_sbc(t, P->z, R->x);            /* T = Z_P − X_R   */
  fp_add(R->x, P->z, R->x);         /* X_R = Z_P + X_R */
  fp_mul(R->z, t, R->x);            /* Z_R = T × X_R   */
  fp_mul(R->y, R->x, R->h);         /* Y_R = X_R × H_R */
  fp_mul(R->x, R->e, t);            /* X_R = E_R × T   */
}

```

```
/**
 * @brief Point doubling.
 *
 * @details
 * Doubling R = 2*P on a twisted Edwards curve with a = -1.
 *
 * @param R Point in extended projective coordinates
 *          [x, y, z, e, h], e*h = t = x*y/z
 * @param P Point in extended projective coordinates
 *          [x, y, z, e, h], e*h = t = x*y/z
 */
void point_dbl(ExtPoint *R, ExtPoint *P)
{
  __m256i t[9];

  fp_sqr(R->e, P->x);               /* E_R = X_P^2 */
  fp_sqr(R->h, P->y);               /* H_R = Y_P^2 */
  fp_sbc(t, R->e, R->h);            /* T = E_R - H_R */
  fp_add(R->h, R->e, R->h);         /* H_R = E_R + H_R */
  fp_add(R->x, P->x, P->y);         /* X_R = X_P + Y_P */
  fp_sqr(R->e, R->x);               /* E_R = X_R^2 */
  fp_sub(R->e, R->h, R->e);         /* E_R = H_R - E_R */
  fp_sqr(R->y, P->z);               /* Y_R = Z_P^2 */
  fp_mul29(R->y, R->y, 2);          /* Y_R = 2 * Y_R */
  fp_add(R->y, t, R->y);            /* Y_R = T + Y_R */
  fp_mul(R->x, R->e, R->y);         /* X_R = E_R * Y_R */
  fp_mul(R->z, R->y, t);            /* Z_R = Y_R * T */
  fp_mul(R->y, t, R->h);            /* Y_R = T * H_R */
}
```