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) execution engines like Intel's Advanced Vector Extensions 2 (AVX2) offer a great potential to accelerate elliptic curve cryptography compared to implementations using only basic x64 instructions. All existing AVX2 implementations of scalar multiplication on e.g. Curve25519 (and alternative curves) are optimized for low latency. We argue in this paper that many real-world applications, such as server-side SSL/TLS handshake processing, would benefit more from throughput-optimized implementations than latency-optimized ones. To support this argument, we introduce a throughput-optimized AVX2 implementation of variable-base scalar multiplication on Curve25519 and fixed-base scalar multiplication on Ed25519. Both implementations perform four scalar multiplications in parallel, where each uses a 64-bit element of a 256-bit vector. The field arithmetic is based on a radix-2^{29} representation of the field elements, which makes it possible to carry out four parallel multiplications modulo a multiple of $p = 2^{255} - 19$ in just 88 cycles on a Skylake CPU. Four variable-base scalar multiplications on Curve25519 require less than 250,000 Skylake cycles, which translates to a throughput of 32,318 scalar multiplications per second at a clock frequency of 2 GHz. For comparison, the to-date best latency-optimized AVX2 implementation has a throughput of some 21,000 scalar multiplications per second on the same Skylake CPU.

Keywords: Throughput-optimized cryptography · Curve25519 · Single instruction multiple data (SIMD) · Advanced vector extension 2 (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 capable 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 AVX-512, 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 expansions in the future. For example, the recently introduced Scalable Vector Extension (SVE) of ARM supports vectors of a length of up to 2048 bits[1] [22], while the RISC-V architecture can have vectors that are even 16,384 bits long [13].

Though originally designed to accelerate audio and video processing, SIMD instruction sets like SSE and AVX turned out to be also beneficial for various kinds of cryptographic algorithms [1, 21]. Using prime-field-based Elliptic Curve Cryptography (ECC) as example, an implementer can take advantage of SIMD parallelism to speed up (i) the field arithmetic by adding or multiplying several limbs of field elements in parallel, (ii) the point addition/doubling by executing e.g. two or four field operations in parallel, and (iii) a combination of both. The latency of arithmetic operations in a large prime field can be reduced with the help of SIMD instructions in a similar way as described in e.g. [6, 11, 12] for the RSA algorithm and other public-key schemes. All these implementations have in common that they employ the product-scanning method [14] in combination with a "reduced-radix" representation (e.g. $w = 28$ bits per limb) to perform multiple-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 numerous possibilities for parallel execution. For example, the so-called ladder-step of the Montgomery ladder for Montgomery curves [19] 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. [9, Algorithm 1] and [15, Fig. 1] for AVX2. Scalar multiplication on twisted Edwards curves [4] can be accelerated through parallel execution at the layer of the point arithmetic as well. For example, 2-way and 4-way parallel implementations of point addition and doubling were introduced in e.g. [5, 7, 10] and [8, 10, 16], respectively; these

[1] SVE registers can be between 128 and 2048 bits long, in steps of 128 bits.

execute either two or four field-arithmetic operations in parallel. Finally, there are also a few implementations that combine parallelism at the field-arithmetic and point-arithmetic layer, which can be characterized 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, thus, uses m elements of a vector. For example, Faz-Hernández et al. developed a (2×2)-way parallel AVX2 implementation of scalar multiplication on Curve25519 and reported an execution time of 121,000 Haswell cycles (or 99,400 Skylake cycles) [10]. Hisil et al. presented very recently an AVX-512 implementation of Curve25519 that is (4×2)-way parallelized (i.e. four field operations in parallel, each of which uses two 64-bit elements) and executes in only 74,368 Skylake cycles [15].

Benchmarking results reported in the recent literature indicate that parallel implementations of Curve25519 do not scale very well when switching from one generation of AVX to the next. While AVX-512 (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 concrete reduction in execution time (i.e. latency) is much smaller, namely around 25% (74,368 vs. 99,400 Skylake cycles [15]). This immediately raises the question of how an implementer can exploit the massive parallelism of future SIMD extensions operating on vectors that may be 2048 bits long, or even longer, given the modest gain achieved in [15]. 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 kinds of application demand a low-latency implementation 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 can argue that a fast implementation of Curve25519 reduces the overall handshake-latency a TLS client experiences when connecting to a server. The main issue with this reasoning is that transmitting the public keys over the Internet will likely introduce an orders-of-magnitude higher latency than the computation of the shared secret. Furthermore, given clock frequencies of 4 GHz, most users will not recognize an execution-time reduction by a few 10,000 cycles. It could now be argued that variable-base scalar multiplication is not only required on the client side, but has to be performed also by the TLS server[2]. Indeed, TLS servers of corporations like Google or Facebook may be confronted with several 10,000 TLS handshakes per second, and a faster Curve25519 implementation will help them 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 in a certain

[2] The termination of SSL/TLS connections is often off-loaded to a so-called "reverse proxy," which transparently translates SSL/TLS sessions to normal TCP sessions for back-end servers. The cryptographic performance of such reverse proxies can be significantly improved with dedicated hardware accelerators. Jang et al. introduced *SSLShader*, a SSL/TLS reverse proxy that uses a Graphics Processing Unit (GPU) to increase the throughput of public-key cryptosystems like RSA [18].

interval. Given this requirement, would it not make sense to optimize software implementations of Curve25519 for maximum throughput instead of minimal latency? What throughput can a throughput-optimized implementation achieve compared to a latency-optimized implementation? Surprisingly, it seems these questions have not yet been answered in the literature[3].

In this paper we make a first step to answer these questions and introduce a throughput-optimized AVX2 implementation of variable-base scalar multiplication on Curve25519 and fixed-base scalar multiplication on Ed25519. Both implementations perform (4×1)-way parallel scalar multiplications; this means they execute four scalar multiplications simultaneously in SIMD fashion, where each can have a different scalar and, in the case of Curve25519, a different base point. The point arithmetic and also the underlying field arithmetic operations 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 fairly easy to implement (by simply vectorizing a reduced-radix implementation for a 32-bit processor), which simplifies the effort for formal verification of the correctness of the software. In addition, we expect this approach to scale well with increasing vector lengths; for example, migrating from AVX2 to AVX-512 should roughly double the throughput. Unlike most previous AVX2 implementations, 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 parallel scalar multiplication when we analyzed different representations, including the classical 25.5 bits-per-limb variant [2]. 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 to-date best latency-optimized AVX2 implementation needs over 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.

2 The AVX2 Instruction Set

Intel's Advanced Vector eXtension 2 (AVX2) is an x86 instruction set extension for SIMD processing that supports packed integer operations on 256-bit wide registers. AVX2 was announced in 2011 and first integrated into the Haswell microarchitecture, which appeared in 2013. Besides the increased length of the integer instructions, AVX2 also differs from the older AVX by the instruction format (i.e. three operands instead of two). We performed our experiments on Haswell and Skylake processors since both were used as reference platforms in previous papers, e.g. [10, 20]. On both the Haswell and Skylake microarchitecture, instructions (including AVX instructions) are fetched from the instruction cache and decoded into micro-operations (micro-ops) by the "front end" of the

[3] A throughput-optimized implementation of variable-base scalar multiplication on a 251-bit *binary* Edwards curve was presented by Bernstein [3]. This implementation uses bitslicing for the low-level binary-field arithmetic and is able to execute 30,000 scalar multiplications per second on an Intel Core 2 Quad Q6600 CPU.

core. These micro-ops are stored in a buffer and will be assigned to available execution ports by the superscalar execution engine. The execution engine can issue micro-ops in an out-of-order fashion, i.e. the order in which the micro-ops are executed is not (necessarily) the order in which they were decoded. This is because micro-ops can leave the buffer and get issued to a suitable execution port as soon as their input-operands are available, even if there are still some older micro-ops in the buffer, which helps to improve processing efficiency. Both the Haswell and the Skylake microarchitecture have a total of eight execution ports (i.e. port 0 to port 7), whereby micro-ops of vector ALU instructions are issued to execution units through port 0, 1, and 5. Memory accesses (i.e. loads and stores) are issued through port 2, 3, 4, and 7, while port 6 handles various kinds of branches. Haswell and Skylake differ regarding the capabilities of the AVX2-related ports (i.e. port 0, 1, 5) and execution engines. Taking the AVX2 instructions `VPMULUDQ`, `VPADDQ`, and `VPAND` as example, these differences can be summarized as follows:

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

The micro-ops of the AVX2 vector multiply instruction `VPMULUDQ` can be issued through two ports on a Skylake CPU, but only one port on a Haswell CPU. As a consequence, the throughput of `VPMULUDQ` differs for these two platforms; it is one instruction/cycle on Haswell, but two instructions/cycle on Skylake.

The operands used by the AVX2 ALU instructions have to be stored in the 256-bit `YMM` vector registers, whereby, depending on the concrete instruction, the two operands are interpreted as vectors consisting of e.g. four 64-bit elements or eight 32-bit elements. Depending on the microarchitecture, several (or even all) operations on the 32 or 64-bit elements are executed in parallel. Similar to other vector units, AVX2 does not support "widening" multiplication, i.e. when using `VPMULUDQ` to multiply vectors of four unsigned 64-bit integers, each of the four products has a length of 64 bits. This, in turn, restricts the length of the limbs of a multiple-precision integer to 32 bits, or even less (e.g. 25–30 bits) in the case of a reduced-radix representation.

3 Vectorized Prime-Field Arithmetic

This section describes our (4×1)-way parallel implementation of arithmetic in $\mathbb{F}_p$ where $p = 2^{255} - 19$. We first introduce the notion of a limb vector set and explain the rationale of its radix-2^{29} representation. Afterward, we demonstrate how limb vector sets can be used to implement (4×1)-way field operations.

3.1 Radix-2^{29} Limb Vector Set

The literature contains numerous discussions on how to represent the elements of the 255-bit prime field $\mathbb{F}_p$ used by Curve25519, whereby the bottom line was

always that the choice of the number representation radix is determined by the characteristics of the target platform [10, 15, 17]. A well-known and widely-used choice is the radix-$2^{25.5}$ representation originally proposed in [2], which means a 255-bit integer consists of ten limbs; five are 25 bits long, while the other five have a length of 26 bits. More formally, a field element f is given as

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

where $0 \leq f_{2j} < 2^{26}$ and $0 \leq f_{2j+1} < 2^{25}$ for $0 \leq j \leq 4$. This representation is attractive because it allows implementers to efficiently integrate the reduction modulo $p = 2^{255} - 19$ into the multiplication due to the fact that $19 f_i$ still fits in a 32-bit integer. In this way, it is possible to delay the propagation of excess bits (often called "carries") from one limb to the next-higher limb until the end of the modular multiplication, which is beneficial since these propagations are highly sequential and, thus, relatively slow on modern CPUs. For example, the (2×2)-way parallel AVX2 implementation for Curve25519 described in [9] uses a radix of $2^{25.5}$ to combine the reduction operation with the multiplication.

A number representation that enables low latency for a (2×2)-way parallel implementation is not necessarily the best choice when high throughout is the main goal. Considering our (4×1)-way strategy and the processing capabilities of the AVX2 engine of Haswell/Skylake, we opted for a radix-2^{29} representation of the field elements, which means any $f \in \mathbb{F}_p$ consists of nine 29-bit limbs:

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

where $0 \leq f_i < 2^{29}$ for $0 \leq i \leq 8$. Our implementation performs all arithmetic operations modulo $q = 2^6 p = 64(2^{255} - 19) = 2^{261} - 1216$ instead of the prime $p = 2^{255} - 19$. A benefit of this representation is the smaller number of limbs compared to the radix-$2^{25.5}$ approach, which usually[4] implies that fewer limb-multiplications have to be carried out when multiplying two field elements. The main drawback is a higher number of excess-bit (i.e. carry) propagations since the reduction modulo q can only be done *after* the multiplication. However, we found through a number of experiments that, on both Haswell and Skylake, the advantage of fewer limbs outweighs the additional carry propagations.

The main data structure of our (4×1)-way parallel software is what we call a *limb vector set*. Given $e, f, g, h \in \mathbb{F}_p$, a limb vector set $\boldsymbol{V}$ is defined as:

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

[4] A (2×2)-way parallel AVX2 implementation (i.e. an implementation executing two field operations in parallel, each using two 64-bit elements of a 256-bit vector) can not profit from a radix-2^{29} representation since the limbs are processed in pairs and the number of limb-pairs is the same as for radix $2^{25.5}$, namely five.

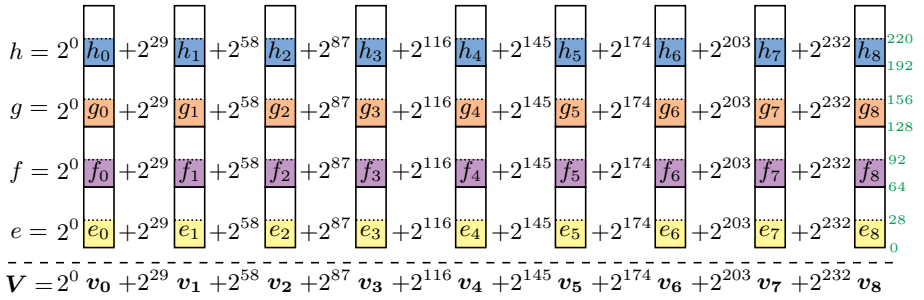

Fig. 1. Limb vector set $\boldsymbol{V}$ representing the four operands $e, f, g, h \in \mathbb{F}_p$ (there are nine limb vectors $\boldsymbol{v_i}$, each of which contains four 29-bit limbs).

In essence, a limb vector set $\boldsymbol{V}$ consists of nine *limb vectors* $\boldsymbol{v_i}$, each of which contains four 29-bit limbs. However, it is important to note that the four limbs in $\boldsymbol{v_i}$ come from four *different* field-elements and have the same index i, i.e. the least-significant limb vector $\boldsymbol{v_0}$ contains the least-significant limbs of $e, f, g, h \in \mathbb{F}_p$, namely e_0, f_0, g_0, and h_0, while $\boldsymbol{v_8}$ contains the four most-significant limbs of e, f, g, and h. In general, the number of limbs per limb vector is determined by the number of elements in a vector of the underlying SIMD engine (four in our case), whereas the number of limb vectors in a limb vector set depends on the bit-length of the prime and the representation radix. Figure 1 shows the structure of an element vector set $\boldsymbol{V}$ for AVX2, where the 29-bit limbs are in four coloured rows (depicting four field-elements), and each column represents a limb vector $\boldsymbol{v_i}$. The exact bit position of a 29-bit limb within a 256-bit AVX2 vector is given on the right of the column of $\boldsymbol{v_8}$; concretely, the four limbs are placed at the bit positions from $64i$ to $64i + 28$ for $0 \leq i \leq 3$. Even though the radix-2^{29} representation does not permit the integration of modular reduction into the multiplication, it provides sufficient "headroom" (namely three bits) to delay the carry propagation of certain operations like the field-addition.

Our representation of operands for high-throughput arithmetic for ECC is similar to the "bit-sliced" and "byte-sliced" representations used in symmetric cryptography (e.g. for DES or AES) to improve the throughput at the expense of latency. Thus, our approach could be referred to as "limb-slicing."

3.2 AVX2 Implementation of Field-Operations

All inputs to the field-operations described in the following are limb vector sets with limbs that are 29 long or slightly longer. As already explained before, the arithmetic operations are performed modulo $q = 64p = 2^{261} - 1216$ and not the actual prime p (except at the very end of a scalar multiplication). An operand can, therefore, be up to 261 bits long. Simplified C source code of some of the (4×1)-way parallel field-operations can be found in Appendix A (except of the (4×1)-way field-multiplication, which is already given in this subsection).

Addition. The vectorized addition $\boldsymbol{R} = \boldsymbol{A} + \boldsymbol{B}$ is implemented in a straightforward way, which means nine `VPADDQ` instructions are executed to obtain the limb-sums $\boldsymbol{r_i} = \boldsymbol{a_i} + \boldsymbol{b_i}$ for $0 \leq i \leq 8$. We neither propagate the carries from less-significant to more-significant limb-vectors, nor do we perform a reduction modulo q. Thus, each limb of the sum vector set $\boldsymbol{R}$ can be one bit longer than the corresponding limbs of the operands $\boldsymbol{A}$ and $\boldsymbol{B}$.

Subtraction. Computing the subtraction in the usual way as $\boldsymbol{R} = \boldsymbol{A} - \boldsymbol{B}$ can yield negative limbs and also negative final results; therefore, we implemented the subtraction using the equation $\boldsymbol{R} = \boldsymbol{2Q} + \boldsymbol{A} - \boldsymbol{B}$, whereby the limb-vectors of $2\boldsymbol{Q}$ have the form $2\boldsymbol{q_i}$, i.e. the limbs are 30 bits long. The limbs of the final result $\boldsymbol{R}$ can be up to 31 bits long, which means they may cause an overflow in the subsequent field-operation (e.g. when the result is used as input of a field-squaring). Therefore, we implemented an alternative version of the subtraction that includes both a carry propagation to obtain 29-bit limbs and a reduction modulo q, which is performed in the usual way (taking $2^{261} \equiv 1216 \bmod q$ into account). The baseline version of the subtraction is very similar to the addition (see above) and executes nine `VPADDQ` and `VPSUBQ` instructions, respectively. On the other hand, the second version of the subtraction is much more costly due to the sequential carry propagation; thus, we use it only when necessary.

Multiplication. Multiplication is (apart from inversion) the most costly field-operation and has a significant impact on the performance of any elliptic-curve cryptosystem. Our AVX2 implementation aims at maximizing instruction-level parallelism by optimizing the port utilization, such that as many micro-ops as possible can be executed simultaneously. However, achieving optimal utilization of ports (and execution units) is not always possible due to inherent sequential dependencies, e.g. when an instruction uses the result of another instruction as operand, or when an instruction has to wait for an operand to be loaded from RAM. But a smart combination of arithmetic algorithms and implementation options can reduce these dependencies, e.g. reduced-radix product scanning is better suited for AVX2 than full-radix operand scanning. Special attention has to paid to the modular reduction and the carry propagation (i.e. conversion to 29-bit limbs) since some sequential dependencies are unavoidable in these operations. In some cases, the length of a dependency chain can be shortened with the help of additional instructions. All this makes finding a multiplication strategy that schedules the instruction sequence to fully exploit the platform's parallel processing capabilities a highly challenging task.

Taking into account the different latency and throughput properties of the relevant AVX2 instructions, we carried out experiments with a dozen variants of modular multiplication; all of them use product-scanning [14] in combination with a reduced-radix representation, but they differ in the following aspects:

1. Whether the reduction modulo q is separated from or interleaved with the multiplication (and, in the latter case, how it is interleaved).

```
#include <immintrin.h>
#define VADD(X,Y) _mm256_add_epi64(X,Y)  /* VPADDQ */
#define VMUL(X,Y) _mm256_mul_epu32(X,Y)  /* VPMULUDQ */
#define VAND(X,Y) _mm256_and_si256(X,Y)  /* VPAND */
#define VSRL(X,Y) _mm256_srli_epi64(X,Y) /* VPSRLQ */
#define VBCAST(X) _mm256_set1_epi64x(X)  /* VPBROADCASTQ */
#define MASK29    0x1fffffff             /* mask of 29 LSBs */

void gfp_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] = VBCAST(0);
    for(j = 0, k = i; k >= 0; j++, k--)
      t[i] = VADD(t[i], VMUL(a[j], b[k]));
  }
  accu = VSRL(t[8], 29);
  t[8] = VAND(t[8], VBCAST(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 = VADD(accu, VMUL(a[j], b[k]));
    r[i-9] = VAND(accu, VBCAST(MASK29));
    accu = VSRL(accu, 29);
  }
  r[8] = accu;

  /* modulo reduction and conversion to 29-bit limbs */
  accu = VBCAST(0);
  for (i = 0; i < 9; i++) {
    accu = VADD(accu, VMUL(r[i], VBCAST(64*19)));
    accu = VADD(accu, t[i]);
    r[i] = VAND(accu, VBCAST(MASK29));
    accu = VSRL(accu, 29);
  }

  /* limbs in r[0] can finally be up to 30 bits long */
  r[0] = VADD(r[0], VMUL(accu, VBCAST(64*19)));
}
```

Listing 1. Simplified C source code for (4×1)-way field-multiplication.

2. In which way the carry propagation is performed.
3. Whether and how intermediate values are stored in local variables.

We benchmarked all 12 variants (including one based on the radix-$2^{25.5}$ representation from [2]) on Haswell and Skylake, and found that the version shown in Listing 1 is the fastest one. This source code implements a (4×1)-way parallel field-multiplication, whereby the reduction modulo q (third loop) is performed separately after the product-scanning multiplication (first two loops). The local array `t` serves as storage for the intermediate results (i.e. column sums) of the first loop; these sums must not exceed 64 bits so as to prevent overflow. In the second loop, the carries are propagated to obtain the column sums in the form of 29-bit limbs (stored in array `r`) since this simplifies the subsequent reduction operation. The reduction of the product modulo q is done in the conventional way (i.e. by using $2^{261} \equiv 1216 \bmod q$) and combined with a carry propagation

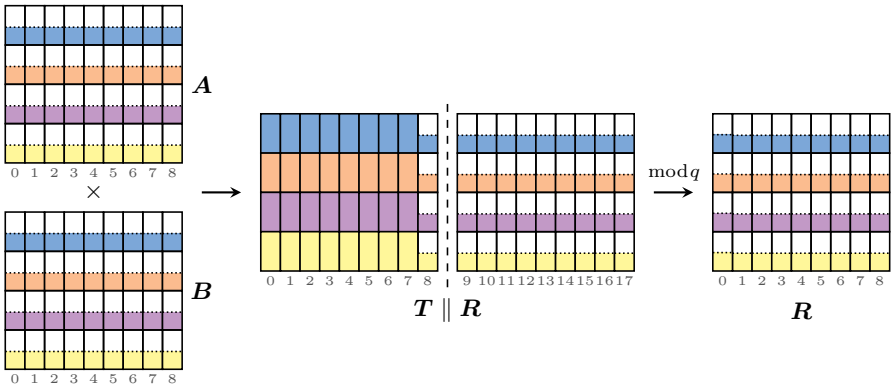

Fig. 2. (4×1)-way field-multiplication ($\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{T}$, and $\boldsymbol{R}$ are limb vector sets).

to get the final result in 29-bit limbs. Consequently, the modular multiplication involves two carry propagations altogether. Figure 2 illustrates our vectorized modular multiplication and shows the limb-length of all 18 column sums of the product $\boldsymbol{AB}$; these sums are stored in the limb vector set $\boldsymbol{T}$ (lower half of the product) and $\boldsymbol{R}$ (upper half), which correspond to `t` and `r` in Listing 1.

An important aspect when choosing a number-representation radix for the product-scanning method is to ensure that the column sums will never become longer than 64 bits so as to prevent overflows. The impossibility of overflows is fairly easy to show when all limbs of the two operands are 29 bits long. In this case, a limb-product consists of 58 bits, and the maximum length of a column sum is 62 bits. The biggest column sum is t_8, which is computed as follows.

$$t_8 = a_0b_8 + a_1b_7 + a_2b_6 + a_3b_5 + a_4b_4 + a_5b_3 + a_6b_2 + a_7b_1 + a_8b_0$$

During the execution of the reduction loop (line 34 to 37 of Listing 1), the sum of t_7, the product $1216r_7$, and the carry of the previous iteration (in `accu`) is computed. This sum can not overflow a 64-bit element of an AVX2 vector since the carry can be at most $64 - 29 = 35$ bits long, and r_7 has a length of 29 bits (i.e. the length of $1216r_7$ is at most 40 bits). It is worth noting that overflows are not possible even when the limbs of the operands are slightly longer than 29 bits. To give a concrete example, let us assume operand $\boldsymbol{A}$ and $\boldsymbol{B}$ are the result of a field-addition and a field-subtraction, respectively. In this case, the limbs are bounded by $a_i < 2 \cdot 2^{29} \leq 2^{30}$ and $b_i < 3 \cdot 2^{29} < 2^{30.59}$, which implies $t_8 < 2^{63.76}$ (i.e. t_8 fits in a 64-bit element of an AVX2 vector). Consequently, an overflow is not possible, neither in the multiplication nor in the reduction.

Squaring. Our implementation of squaring $\boldsymbol{R} = \boldsymbol{A}^2 \bmod \boldsymbol{Q}$ exploits the usual "shortcut" of computing limb-products of the form $\boldsymbol{a_i a_j}$ with $i \neq j$ once and doubling them by a left-shift. Apart from that, we applied similar optimization strategies with respect to carry propagation and reduction as outlined above.

4 (4×1)-Way Parallel Scalar Multiplication

An ephemeral ECDH key exchange requires both variable-base and fixed-base scalar multiplication; the latter to generate a key pair and the former to obtain the shared secret. Our implementation adopts Curve25519 for the computation of shared secrets and Ed25519 to generate key pairs, whereby the public keys need to be mapped to Curve25519. Both scalar multiplications are vectorized in a (4×1)-way parallel fashion, i.e. four scalar multiplications are carried out in parallel, whereby they can use four different scalars (and also four different base points in the variable-base setting).

4.1 Variable-Base Scalar Multiplication

The Montgomery ladder [19] is the standard way to implement a variable-base scalar multiplication kP on Curve25519. For each bit of the scalar k, a so-called *ladder step* is performed, which mainly consists of a differential point addition and a point doubling; both operations can be carried out with the (projective) X and Z coordinate only (i.e. the Y coordinate is not needed). The ladder has constant run-time since each step executes a fixed instruction sequence.

Point Vector Set. A *point vector set* consists of several limb vector sets (one for each coordinate), which represents always four points, corresponding to the number of limbs in a limb vector set. For example, a point vector set in affine coordinates for the four points $A = (x_A, y_A)$, $B = (x_B, y_B)$, $C = (x_C, y_C)$, and $D = (x_D, y_D)$ can be written as

$$\begin{aligned}\boldsymbol{\mathcal{P}} = [A, B, C, D] &= [(x_A, y_A), (x_B, y_B), (x_C, y_C), (x_D, y_D)] = \\ &= ([x_A, x_B, x_C, x_D], [y_A, y_B, y_C, y_D]) = (\boldsymbol{x_\mathcal{P}}, \boldsymbol{y_\mathcal{P}}).\end{aligned}$$

That is, $\boldsymbol{\mathcal{P}}$ consists of the two limb vector sets $\boldsymbol{x_\mathcal{P}}$ and $\boldsymbol{y_\mathcal{P}}$; the former contains the four x-coordinates of A, B, C, D, and the latter the four y-coordinates. In the case of projective coordinates, $\boldsymbol{\mathcal{P}}$ has the form $\boldsymbol{\mathcal{P}} = (\boldsymbol{X_\mathcal{P}}, \boldsymbol{Y_\mathcal{P}}, \boldsymbol{Z_\mathcal{P}})$.

(4×1)-Way Montgomery Ladder. The normal ladder-step operation gets the (projective) X and Z coordinate of two points P and Q as input, plus the affine x-coordinate x_{QP} of the difference $Q - P$ of the points (which actually is a public key). It outputs two points P' and Q' whose difference $Q' - P'$ has the same affine x-coordinate as $Q - P$. However, our (4×1)-way implementation of the ladder step operates on point vector sets instead of ordinary points. To simplify the explanation, we write the conventional Montgomery ladder step as $(P', Q') \leftarrow \text{LStep}(P, Q, x_{QP})$, and, analogously, the (4×1)-way ladder step as $(\boldsymbol{\mathcal{P}'}, \boldsymbol{\mathcal{Q}'}) \leftarrow \text{ParLStep}(\boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{Q}}, \boldsymbol{x_{\mathcal{QP}}})$. We suppose that the two point vector sets $\boldsymbol{\mathcal{P}}$ and $\boldsymbol{\mathcal{Q}}$ represent the points A, B, C, D and E, F, G, H, respectively, whereas the limb vector set $\boldsymbol{x_{\mathcal{QP}}}$ represents the field-elements $x_{EA}, x_{FB}, x_{GC}, x_{HD}$, all of which are affine x-coordinates of differences of two points (these coordinates

Op.	ParLStep	Instance 0	Instance 1	Instance 2	Instance 3
1	$\boldsymbol{T} \leftarrow \boldsymbol{X_\mathcal{P}} + \boldsymbol{Z_\mathcal{P}}$	$s \leftarrow X_A + Z_A$	$t \leftarrow X_B + Z_B$	$u \leftarrow X_C + Z_C$	$v \leftarrow X_D + Z_D$
2	$\boldsymbol{X_\mathcal{P}} \leftarrow \boldsymbol{X_\mathcal{P}} - \boldsymbol{Z_\mathcal{P}}$	$X_A \leftarrow X_A - Z_A$	$X_B \leftarrow X_B - Z_B$	$X_C \leftarrow X_C - Z_C$	$X_D \leftarrow X_D - Z_D$
3	$\boldsymbol{T'} \leftarrow \boldsymbol{X_\mathcal{Q}} + \boldsymbol{Z_\mathcal{Q}}$	$s' \leftarrow X_E + Z_E$	$t' \leftarrow X_F + Z_F$	$u' \leftarrow X_G + Z_G$	$v' \leftarrow X_H + Z_H$
4	$\boldsymbol{X_\mathcal{Q}} \leftarrow \boldsymbol{X_\mathcal{Q}} - \boldsymbol{Z_\mathcal{Q}}$	$X_E \leftarrow X_E - Z_E$	$X_F \leftarrow X_F - Z_F$	$X_G \leftarrow X_G - Z_G$	$X_H \leftarrow X_H - Z_H$
5	$\boldsymbol{Z_\mathcal{P}} \leftarrow \boldsymbol{T}^2$	$X_A \leftarrow s^2$	$X_B \leftarrow t^2$	$X_C \leftarrow u^2$	$X_D \leftarrow v^2$
6	$\boldsymbol{Z_\mathcal{Q}} \leftarrow \boldsymbol{T'} \times \boldsymbol{X_\mathcal{P}}$	$Z_E \leftarrow s' \times X_A$	$Z_F \leftarrow t' \times X_B$	$Z_G \leftarrow u' \times X_C$	$Z_H \leftarrow v' \times X_D$
7	$\boldsymbol{T'} \leftarrow \boldsymbol{X_\mathcal{Q}} \times \boldsymbol{T}$	$s' \leftarrow X_E \times s$	$t' \leftarrow X_F \times t$	$u' \leftarrow X_G \times u$	$v' \leftarrow X_H \times v$
8	$\boldsymbol{T} \leftarrow \boldsymbol{X_\mathcal{P}}^2$	$s \leftarrow X_A{}^2$	$t \leftarrow X_B{}^2$	$u \leftarrow X_C{}^2$	$v \leftarrow X_D{}^2$
9	$\boldsymbol{X_\mathcal{P}} \leftarrow \boldsymbol{Z_\mathcal{P}} \times \boldsymbol{T}$	$X_A \leftarrow Z_A \times s$	$X_B \leftarrow Z_B \times t$	$X_C \leftarrow Z_C \times u$	$X_D \leftarrow Z_D \times v$
10	$\boldsymbol{T} \leftarrow \boldsymbol{Z_\mathcal{P}} - \boldsymbol{T}$	$s \leftarrow Z_A - s$	$t \leftarrow Z_B - t$	$u \leftarrow Z_C - u$	$v \leftarrow Z_D - v$
11	$\boldsymbol{X_\mathcal{Q}} \leftarrow \boldsymbol{T} \times \boldsymbol{a_{24}}$	$X_E \leftarrow s \times a_{24}$	$X_F \leftarrow t \times a_{24}$	$X_G \leftarrow u \times a_{24}$	$X_H \leftarrow v \times a_{24}$
12	$\boldsymbol{X_\mathcal{Q}} \leftarrow \boldsymbol{X_\mathcal{Q}} + \boldsymbol{Z_\mathcal{P}}$	$X_E \leftarrow X_E + Z_A$	$X_F \leftarrow X_F + Z_B$	$X_G \leftarrow X_G + Z_C$	$X_H \leftarrow X_H + Z_D$
13	$\boldsymbol{Z_\mathcal{P}} \leftarrow \boldsymbol{X_\mathcal{Q}} \times \boldsymbol{T}$	$Z_A \leftarrow X_E \times s$	$Z_B \leftarrow X_F \times t$	$Z_C \leftarrow X_G \times u$	$Z_D \leftarrow X_H \times v$
14	$\boldsymbol{T} \leftarrow \boldsymbol{T'} + \boldsymbol{Z_\mathcal{Q}}$	$s \leftarrow s' + Z_E$	$t \leftarrow t' + Z_F$	$u \leftarrow u' + Z_G$	$v \leftarrow v' + Z_H$
15	$\boldsymbol{X_\mathcal{Q}} \leftarrow \boldsymbol{T}^2$	$X_E \leftarrow s^2$	$X_F \leftarrow t^2$	$X_G \leftarrow u^2$	$X_H \leftarrow v^2$
16	$\boldsymbol{T} \leftarrow \boldsymbol{T'} - \boldsymbol{Z_\mathcal{Q}}$	$s \leftarrow s' - Z_E$	$t \leftarrow t' - Z_F$	$u \leftarrow u' - Z_G$	$v \leftarrow v' - Z_H$
17	$\boldsymbol{T'} \leftarrow \boldsymbol{T}^2$	$s' \leftarrow s^2$	$t' \leftarrow t^2$	$u' \leftarrow u^2$	$v' \leftarrow v^2$
18	$\boldsymbol{Z_\mathcal{Q}} \leftarrow \boldsymbol{T'} \times \boldsymbol{x_{\mathcal{QP}}}$	$Z_E \leftarrow s' \times x_{EA}$	$Z_F \leftarrow t' \times x_{FB}$	$Z_G \leftarrow u' \times x_{GC}$	$Z_H \leftarrow v' \times x_{HD}$

Fig. 3. (4×1)-way parallel Montgomery ladder step. This implementation works "in place" and updates the input $\boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{Q}}$ with the output $\boldsymbol{\mathcal{P}'}, \boldsymbol{\mathcal{Q}'}$. The subtractions at step 2 and 16 require a reduction and conversion of the result to 29-bit limbs.

are, in fact, public keys). The relation between ParLStep and LStep can be formally described as follows:

$$\begin{aligned}
&\textsc{ParLStep}(\boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{Q}}, \boldsymbol{x_{\mathcal{QP}}}) = \\
&\textsc{ParLStep}([A,B,C,D],[E,F,G,H],[x_{EA},x_{FB},x_{GC},x_{HD}]) = \\
&[\textsc{LStep}(A,E,x_{EA}), \textsc{LStep}(B,F,x_{FB}), \textsc{LStep}(C,G,x_{GC}), \textsc{LStep}(D,H,x_{HD})]
\end{aligned}$$

ParLStep consists of four LStep operations, which are carried out in parallel and execute (besides other operations) five (4×1)-way field-multiplications as well as four (4×1)-way field-squarings. Figure 3 illustrates the exact sequence of field-operations for each of the four parallel LStep instances. There are two temporary limb vector sets $\boldsymbol{T} = [s,t,u,v]$ and $\boldsymbol{T'} = [s',t',u',v']$ involved in the computation, and also the limb vector $\boldsymbol{a_{24}} = [a_{24}, a_{24}, a_{24}, a_{24}]$, which contains four times the curve constant $a_{24} = (A-2)/4 = 121665$.

4.2 Fixed-Base Scalar Multiplication

The fixed-base scalar multiplication $R = kG$ is performed on Ed25519, which is a twisted Edwards curve that is birationally equivalent to Curve25519 [5]. The generator G has the form $(x, 4/5)$ (corresponding to the generator $G = (9, y)$ on Curve25519) and the scalar k is a 255-bit integer. This scalar can be written as $\sum_{i=0}^{63} 16^i k_i$ where $k_i \in \{-8, -7, \ldots, 6, 7\}$ and $R = kG$ computed via

$$R = \sum_{i=0}^{63} k_i \cdot 16^i G. \tag{2}$$

Since the point G is fixed, it is possible to speed up the scalar multiplication with the help of a pre-computed look-up table as demonstrated in e.g. [5]. The table contains the eight multiples 16^iG, $2 \cdot 16^iG$, ..., $8 \cdot 16^iG$ for each of the 64 points of the form 16^iG, which amounts to 512 points altogether. One can efficiently obtain $k_i \cdot 16^iG$ by first loading the point $|k_i| \cdot 16^iG$ from the table and then negating it when $k_i < 0$. Both the table look-up and the conditional negation can be implemented in such a way that they do not leak information about the secret scalar in a timing attack; see [5] for details. It is also possible to reduce the size of the look-up table by splitting Eq. (2) into two parts:

$$R = \sum_{i=0}^{31} k_{2i} \cdot 16^{2i}G + 16 \cdot \sum_{i=0}^{31} k_{2i+1} \cdot 16^{2i}G. \tag{3}$$

In this way, the size of the look-up table is reduced from $64 \times 8 = 512$ points to $32 \times 8 = 256$ points at the expense of four point doublings. The full cost of the fixed-base scalar multiplication according to Eq. (3) amounts to 64 table look-ups, 64 point additions, and four point doublings. Our AVX2 implementation uses this method and performs four scalar multiplications in parallel via

$$\boldsymbol{\mathcal{R}} = \sum_{i=0}^{31} \boldsymbol{k_{2i}} \cdot 16^{2i}\boldsymbol{\mathcal{G}} + 16 \cdot \sum_{i=0}^{31} \boldsymbol{k_{2i+1}} \cdot 16^{2i}\boldsymbol{\mathcal{G}}, \tag{4}$$

where $\boldsymbol{\mathcal{G}} = [G, G, G, G]$, i.e. each instance uses the same generator. The table in our software does not need to be a vectorized table containing four duplicate entries. Instead, we use a normal table to load the four points corresponding to nibbles of the four scalars and generate a point vector set with them.

Look-Up Table. As mentioned above, the pre-computed look-up table holds 256 points in total, all of which are multiples of the fixed generator G. We use a *full-radix* representation instead of the radix-2^{29} representation for the points in the table, i.e. the limbs of the coordinates are 32 bits long. Furthermore, we store the points in *extended affine coordinates* of the form (u, v, w) where

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

and d is a parameter of the twisted Edwards curve [5]. Hence, each coordinate occupies 32 bytes in the table and a point has a size of 96 bytes. The total size of the look-up table is roughly 24 kB. Representing the pre-computed points in extended projective coordinates allows one to use the highly-efficient formulae for mixed point addition presented in [16]. In order to resist timing attacks, all eight points of each set of the form $\{ 16^{2i}G, 2 \cdot 16^{2i}G, \ldots, 8 \cdot 16^{2i}G \}$ need to be loaded from the table and the desired point has to obtained through arithmetic (resp. logical) operations as described in [5]. However, our (4×1)-way parallel software actually obtains four points, one for each of the four scalars. Once the four points have been extracted, a point vector set is generated, which includes a conversion of coordinates from full-radix to reduced-radix representation.

Table 1. Execution time (in cycles) of (4×1)-way field and point operations.

Domain	Operation	Faz-H. et al. [10]		This work	
		Haswell	Skylake	Haswell	Skylake
Prime field $\mathbb{F}_p$	Addition	12	12	11	11
	Basic subtraction	n/a	n/a	14	12
	Mod. subtraction	n/a	n/a	32	31
	Mod. multiplication	159	105	122	88
	Mod. 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	n/a	n/a	1118	818

(4×1)-Way Point Operations. A (4×1)-way parallel scalar multiplication in the fixed-base setting consists of three kinds of operation: table query, mixed point addition, and point doubling. The latter two operate on point vector sets and execute four instances in parallel, using the AVX2 implementations of the field arithmetic. All three operations have in common that they do not perform any secret-dependent memory accesses or branches, which makes our software highly resistant against timing attacks. The C source code of the point addition and point doubling can be found in Appendix B.

5 Performance Evaluation and Comparison

We measured the execution time of our software on two 64-bit Intel processors that come with an AVX2 engine; the first is a Core i7-4710HQ Haswell clocked with a clock frequency of 2.5 GHz, and the second is a Core i5-6360U Skylake clocked at 2.0 GHz. The source codes were compiled with Clang version 10.0.0 on both processors, using optimization level O2. We disabled turbo boost and hyper-threading during the performance measurements.

Table 1 shows the latency of our (4×1)-way parallel arithmetic operations in the prime field and on the two curves, and compares them with the (4×1)-way operations reported by Faz-Hernández et al. in [10]. They implemented the modular multiplication based on a number representation radix of $2^{25.5}$ so as to minimize the carry propagation. However, according to the results given in Table 1, our 29-bits-per-limb multiplication outperforms [10] by 37 clock cycles on Haswell and 17 cycles on Skylake, despite the drawback of requiring more carry propagations. Also our squaring is faster than that from [10], and these gains at the field arithmetic translate to faster point arithmetic; e.g. they make point addition on Ed25519 exactly 101 cycles faster on Haswell and 135 cycles faster on Skylake. We also benchmarked point operations on Curve25519; the ladder step takes 1118 cycles on Haswell and 300 cycles less on Skylake.

Table 2. Latency and throughput of our AVX2 software on a Haswell i7-4710HQ and a Skylake i5-6360U CPU (latency is the execution time of four parallel instances).

Platform	CPU Frequency	Keypair generation Latency	Keypair generation Throughput	Shared secret Latency	Shared secret Throughput
Haswell	2.5 GHz	104,579 cycles	95,568 ops/sec	329,455 cycles	30,336 ops/sec
Skylake	2.0 GHz	80,249 cycles	99,363 ops/sec	246,636 cycles	32,318 ops/sec

Table 2 shows the performance of our (4×1)-way parallel implementation of keypair generation (i.e. fixed-base scalar multiplication on Ed25519) and the computation of shared secret keys (i.e. variable-base scalar multiplication on Curve25519). The latency figures represent the execution time of four parallel scalar multiplications; for example, computing four variable-base scalar multiplications takes 246.6 k cycles on Skylake. Thanks to a 24 kB look-up table, the generation of keypairs is (on both platforms) over three times faster than the computation of shared secrets. Regarding throughput, our software is able to compute 95 k keypairs or 30 k shared secrets on a 2.5 GHz Haswell CPU. When clocked with the same frequency, a Skylake CPU running our software reaches a 30% higher throughout than a Haswell CPU.

Table 3. Comparison of AVX2 implementations of Curve25519 on Haswell CPUs (all throughput figures are scaled to a common clock frequency of 2.5 GHz).

Ref.	Impl.	CPU	Compiler	Keypair generation Latency [cycles]	Keypair generation Throughput [ops/sec]	Shared secret Latency [cycles]	Shared secret Throughput [ops/sec]
Faz-H.	(2×2)-way	i7-4770	Clang 5.0.2	43,700	57,208†	121,000	20,661†
[10]	(2×2)-way	i7-4710HQ	Clang 10.0	41,938	59,575	121,499	20,563
Nath	(4×1)-way	i7-6500U	GCC 7.3.0	100,127	24,968†	120,108	20,815†
[20]	(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	104,579⋆	**95,568** +60.4%	329,455⋆	**30,336** +45.7%

† Reference [10] and [20] do not give throughput results. Therefore, we list the theoretical throughput obtained by dividing the frequency of 2.5 GHz by the latency (in cycles).
⋆ The latency of our implementation is the execution time of four parallel instances.

Table 3 compares our results with that of two other AVX2 implementations of Curve25519 on the Haswell microarchitecture. In order to reduce (as much as possible) the impact of different compilers and different CPUs, we downloaded and compiled the source code of [10] and [20] in our own experimental environment and measured the corresponding performance data. The results given in Table 3 include both our own measurements and the results which the authors reported in the respective papers. The former are easy to identify in the table

Table 4. Comparison of AVX2 implementations of Curve25519 on Skylake CPUs (all throughput figures are scaled to a common clock frequency of 2.0 GHz).

Ref.	Impl.	CPU	Compiler	Keypair generation Latency [cycles]	Keypair generation Throughput [ops/sec]	Shared secret Latency [cycles]	Shared secret Throughput [ops/sec]
Faz-H.	(2×2)-way	i7-6700K	Clang 5.0.2	34,500	57,971‡	99,400	20,150‡
[10]	(2×2)-way	i5-6360U	Clang 10.0	35,629	55,955	95,129	20,939
Hisil	(4×1)-way	i9-7900X	GCC 5.4	n/a	n/a	98,484	20,308†
[15]	(4×1)-way	i5-6360U	GCC 8.4.0	n/a	n/a	116,595	16,656
Nath	(4×1)-way	i7-6500U	GCC 7.3.0	84,047	23,796†	95,437	20,956†
[20]	(4×1)-way	i5-6360U	GCC 8.4.0	82,054	24,406	93,657	21,168
This	(4×1)-way	i5-6360U	Clang 10.0	80,249⋆	**99,363**	246,636⋆	**32,318**
work					+71.4%		+52.7%

† Reference [15] and [20] do not give throughput results. Therefore, we list the theoretical throughput obtained by dividing the frequency of 2.0 GHz by the latency (in cycles).
‡ Reference [10] provides throughput results for a 3.6 GHz CPU. We list the theoretical throughput obtained by scaling the reported throughput with a factor of $2/3.6 = 1/1.8$ to facilitate an intuitive comparison of the results.
⋆ The latency of our implementation is the execution time of four parallel instances.

because we used more recent versions of the compiler (GCC or Clang) and also a newer Haswell CPU for benchmarking. We tried to compile Nath et al.'s code with Clang version 10.0.0, but the performance was worse than that reported in [20]. A possible explanation might be that Nath et al. "tuned" their source code specifically for the capabilities of GCC, which often comes at the expense of sub-optimal performance when using a different compiler. In order to ensure a fair comparison, we compiled their source code with the most recent version of GCC, namely GCC 8.4.0 (released in March 2020). Since throughput figures depend on the clock frequency of the CPU, we "normalized" the throughputs in Table 3 for a frequency of 2.5 GHz; this makes it also easier to compare the three implementations. Besides measuring the throughput, it is, of course, also possible to obtain theoretical throughput values by simply dividing the CPU's clock frequency by the latency-cycles of a single instance. The implementation introduced in [10] performs (2×2)-way field operations, but it takes advantage of a (4×1)-way table query for fixed-base scalar multiplication. Our software outperforms [10] in terms of throughput of keypair generation by over 60%. On the other hand, the throughput of Nath et al. [20] is much lower since they do not employ a look-up table in the fixed-base scenario. As for the computation of shared secrets, our implementation achieves a 45.7% higher throughput than the currently best latency-optimized software, which is that of [20].

A comparison between our software and other implementations on the Skylake platform can be found in Table 4. Similar as before, we include both the original results reported by the authors and the latency/throughput measured by ourselves. Our implementation is able to generate almost 100,000 key pairs

per second on a 2.0 GHz Skylake CPU, which exceeds the throughput achieved in [10] by roughly 71.4%. The currently fastest latency-optimized variable-base scalar multiplication was introduced by Nath et al. in [20]; the execution time of their implementation became even slightly better (by 1780 cycles) when we compiled it with the latest version of GCC. Nonetheless, our implementation outperforms throughput-wise the software of Nath et al. by 52.7%. The results in Table 3 and Table 4 indicate that using a newer version of a compiler does not always yield better performance. For example, Hisil et al.'s implementation for Skylake described in [15] became actually slower when the source code was compiled with GCC 8.4.0 instead of GCC 5.4.

6 Conclusions

The length of vectors supported by common vector instruction sets like Intel's AVX has increased steadily over the past ten years, and this trend is expected to continue. For example, future generations of ARM and RISC-V processors can use vectors that are more than 1000 bits long. This makes a strong case to research how such enormous parallel processing capabilities can be exploited in ECC. An analysis of existing AVX implementations of Curve25519, which are all optimized for low latency, indicates that they will most likely not be able to gain much from longer vectors and increased vector parallelism, mainly due to some inherent sequential dependencies in the point arithmetic. We proposed in this paper to use vector engines to maximize throughput rather than minimize latency. In particular, we introduced the "limb-slicing" technique for AVX2 to execute four scalar multiplications with different inputs in parallel, each using a 64-bit element of a 256-bit vector. The parallel processing of four instances of scalar multiplication that are fully independent among each other improves the utilization of the vector engine and increases throughput. Our experiments confirm that limb-sliced AVX2 implementations of fixed-base and variable-base scalar multiplication outperform their latency-optimized counterparts in terms of throughput by up to 71.4%. Furthermore, limb-slicing also scales very well across vector lengths; for example, one can expect that migrating our software from AVX2 to AVX-512 will roughly double the throughput.

We envision that limb-slicing will play a similar role in public-key cryptography as bit-slicing in symmetric cryptography and hope that this paper serves as inspiration for future research activities in high-performance cryptographic software. As part of our future work we plan to develop a limb-sliced software implementation of EdDSA signature verification using AVX-512 instructions. In addition, we will develop high-throughput software for isogeny-based ECC.

Acknowledgements. The source code of the presented software is available online at `https://gitlab.uni.lu/APSIA/AVXECC` under GPLv3 license. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM).

References

1. K. Aoki, F. Hoshino, T. Kobayashi, and H. Oguro. Elliptic curve arithmetic using SIMD. In G. I. Davida and Y. Frankel, editors, *Information Security — ISC 2001*, volume 2200 of *Lecture Notes in Computer Science*, pages 235–247. Springer Verlag, 2001.
2. 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.
3. D. J. Bernstein. Batch binary Edwards. In S. Halevi, editor, *Advances in Cryptology — CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 317–336. Springer Verlag, 2009.
4. 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.
5. 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.
6. J. W. Bos, P. L. Montgomery, D. Shumow, and G. M. Zaverucha. Montgomery multiplication using vector instructions. In T. Lange, K. Lauter, and P. Lisonek, editors, *Selected Areas in Cryptography — SAC 2013*, volume 8282 of *Lecture Notes in Computer Science*, pages 471–489. Springer Verlag, 2014.
7. T. Chou. Sandy2x: New Curve25519 speed records. In O. Dunkelman and L. Keliher, editors, *Selected Areas in Cryptography — SAC 2015*, volume 9566 of *Lecture Notes in Computer Science*, pages 145–160. Springer Verlag, 2016.
8. H. de Valence. Accelerating Edwards curve arithmetic with parallel formulas. Blog post, available online at `https://medium.com/@hdevalence/accelerating-edwards-curve-arithmetic-with-parallel-formulas-ac12cf5015be`, 2018.
9. 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.
10. A. Faz-Hernández, J. López, and R. Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Transactions on Mathematical Software*, 45(3):1–35, July 2019.
11. 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.
12. 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.
13. T. R. Halfhill. RISC-V vectors know no limits. Linley Newsletter, available online at `https://www.linleygroup.com/newsletters/newsletter_detail.php?num=6154`, 2020.
14. D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.

15. H. Hişil, B. Eğrice, and M. Yassi. Fast 4 way vectorized ladder for the complete set of Montgomery curves. Cryptology ePrint Archive, Report 2020/388, 2020. Available for download at `https://eprint.iacr.org`.
16. 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.
17. J. Huang, Z. Liu, Z. Hu, and J. Großschädl. Parallel implementation of SM2 elliptic curve cryptography on Intel processors with AVX2. In J. K. Liu and H. Cui, editors, *Information Security and Privacy — ACISP 2020*, volume 12248 of *Lecture Notes in Computer Science*, pages 204–224. Springer Verlag, 2020.
18. K. Jang, S. Han, S. Han, S. B. Moon, and K. Park. SSLShader: Cheap SSL acceleration with commodity processors. In D. G. Andersen and S. Ratnasamy, editors, *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2011)*. USENIX Association, 2011.
19. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, Jan. 1987.
20. K. Nath and P. Sarkar. Efficient 4-way vectorizations of the Montgomery ladder. Cryptology ePrint Archive, Report 2020/378, 2020. Available for download at `https://eprint.iacr.org`.
21. D. Page and N. P. Smart. Parallel cryptographic arithmetic using a redundant Montgomery representation. *IEEE Transactions on Computers*, 53(11):1474–1482, Nov. 2004.
22. N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Prémillieu, A. Reid, A. Rico, and P. Walker. The ARM scalable vector extension. *IEEE Micro*, 37(2):26–39, March/April 2017.

A Source Code of Vectorized Field Operations

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

```
#include <immintrin.h>
#define VADD(X,Y) _mm256_add_epi64(X,Y)  /* VPADDQ */
#define VSUB(X,Y) _mm256_sub_epi64(X,Y)  /* VPSUBQ */
#define VMUL(X,Y) _mm256_mul_epu32(X,Y)  /* VPMULUDQ */
#define VAND(X,Y) _mm256_and_si256(X,Y)  /* VPAND */
#define VSRL(X,Y) _mm256_srli_epi64(X,Y) /* VPSRLQ */
#define VSLL(X,Y) _mm256_slli_epi64(X,Y) /* VPSLLQ */
#define VBCAST(X) _mm256_set1_epi64x(X)  /* VPBROADCASTQ */
#define MASK29    0x1fffffff             /* mask of 29 LSBs */

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

/* field subtraction (without a carry propagation) */
void gfp_sub(__m256i *r, const __m256i *a, const __m256i *b)
{
  /* subtraction loop */
  r[0] = VADD(VBCAST(2*0x1ffffb40), VSUB(a[0], b[0]);
  for (int i = 1; i < 9; i++)
    r[i] = VADD(VBCAST(2*0x1fffffff), VSUB(a[i], b[i]);
}
```

```
25
26 /* field subtraction (with a carry propagation) */
27 void gfp_sbc(__m256i *r, const __m256i *a, const __m256i *b)
28 {
29   /* subtraction loop */
30   r[0] = VADD(VBCAST(2*0x1ffffb40), VSUB(a[0], b[0]);
31   for (int i = 1; i < 9; i++)
32     r[i] = VADD(VBCAST(2*0x1fffffff), VSUB(a[i], b[i]);
33
34   /* carry propagation and conversion to 29-bit limbs*/
35   for (int i = 1; i < 9; i++) {
36     r[i] = VADD(r[i], VSRL(r[i-1], 29));
37     r[i-1] = VAND(r[i-1], VBCAST(MASK29));
38   }
39
40   /* limbs in r[0] can finally be 30 bits long */
41   r[0] = VADD(r[0], VMUL(VBCAST(64*19), VSRL(r[8], 29)));
42   r[8] = VAND(r[8], VBCAST(MASK29));
43 }
44
45 /* field squaring */
46 void gfp_sqr(__m256i *r, const __m256i *a)
47 {
48   int i, j, k; __m256i t[9], accu, temp;
49
50   /* 1st loop of the product-scanning squaring */
51   t[0] = VMUL(a[0],a[0]);
52   for (i = 1; i < 9; i++) {
53     t[i] = VBCAST(0);
54     for (j = 0, k = i; j < k; j++, k--)
55       t[i] = VADD(t[i], VMUL(a[j], a[k]));
56     t[i] = VSLL(t[i], 1);
57     if (!(i&1)) t[i] = VADD(t[i], VMUL(a[j], a[j]));
58   }
59   accu = VSRL(t[8], 29);
60   t[8] = VAND(t[8], VBCAST(MASK29));
61
62   /* 2nd loop of the product-scanning squaring */
63   for (i = 9; i < 16; i++) {
64     temp = VBCAST(0);
65     for (j = i-8, k = 8; j < k; j++, k--)
66       temp = VADD(r[i-9], VMUL(a[j], a[k]));
67     accu = VADD(accu, VSLL(temp, 1));
68     if (!(i&1)) accu = VADD(accu, VMUL(a[j], a[j]));
69     r[i-9] = VAND(accu, VBCAST(MASK29));
70     accu = VSRL(accu, 29);
71   }
72   accu = VADD(accu, VMUL(a[8], a[8]));
73   r[7] = VAND(accu, VBCAST(MASK29));
74   r[8] = VSRL(accu, 29);
75
76   /* modulo reduction and conversion to 29-bit limbs */
77   accu = VBCAST(0);
78   for (i = 0; i < 9; i++){
79     accu = VADD(accu, VMUL(r[i], VBCAST(64*19))));
80     accu = VADD(accu, t[i]);
81     r[i] = VAND(accu, VBCAST(MASK29));
82     accu = VSRL(accu, 29);
83   }
84
85   /* limbs in r[0] can finally be 30 bits long */
86   r[0] = ADD(r[0], VMUL(accu, VBCAST(64*19)));
87 }
```

B Source Code of (4 × 1)-Way Point Operations

Listing 3. Simplified 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];

  gfp_mul(t, P->e, P->h);               /*   T = E_P × H_P   */
  gfp_sub(R->e, P->y, P->x);            /*   E_R = Y_P − X_P   */
  gfp_add(R->h, P->y, P->x);            /*   H_R = Y_P + X_P   */
  gfp_mul(R->x, R->e, Q->y);            /*   X_R = E_R × Y_Q   */
  gfp_mul(R->y, R->h, Q->x);            /*   Y_R = H_R × X_Q   */
  gfp_sub(R->e, R->y, R->x);            /*   E_R = Y_R − X_R   */
  gfp_add(R->h, R->y, R->x);            /*   H_R = Y_R + X_R   */
  gfp_mul(R->x, t, Q->z);               /*   X_R = T × Z_Q   */
  gfp_sbc(t, P->z, R->x);               /*   T = Z_P − X_R   */
  gfp_add(R->x, P->z, R->x);            /*   X_R = Z_P + X_R   */
  gfp_mul(R->z, t, R->x);               /*   Z_R = T × X_R   */
  gfp_mul(R->y, R->x, R->h);            /*   Y_R = X_R × H_R   */
  gfp_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];

  gfp_sqr(R->e, P->x);                  /*   E_R = X_P^2   */
  gfp_sqr(R->h, P->y);                  /*   H_R = Y_P^2   */
  gfp_sbc(t, R->e, R->h);               /*   T = E_R − H_R   */
  gfp_add(R->h, R->e, R->h);            /*   H_R = E_R + H_R   */
  gfp_add(R->x, P->x, P->y);            /*   X_R = X_P + Y_P   */
  gfp_sqr(R->e, R->x);                  /*   E_R = X_R^2   */
  gfp_sub(R->e, R->h, R->e);            /*   E_R = H_R − E_R   */
  gfp_sqr(R->y, P->z);                  /*   Y_R = Z_P^2   */
  gfp_mul29(R->y, R->y, 2);             /*   Y_R = 2 · Y_R   */
  gfp_add(R->y, t, R->y);               /*   Y_R = T + Y_R   */
  gfp_mul(R->x, R->e, R->y);            /*   X_R = E_R × Y_R   */
  gfp_mul(R->z, R->y, t);               /*   Z_R = Y_R × T   */
  gfp_mul(R->y, t, R->h);               /*   Y_R = T × H_R   */
}
```