UNIVERSIDADE ESTADUAL DE CAMPINAS
SISTEMA DE BIBLIOTECAS DA UNICAMP
REPOSITÓRIO DA PRODUÇÃO CIENTIFICA E INTELECTUAL DA UNICAMP

Versão do arquivo anexado / Version of attached file:

Versão do Editor / Published Version

Mais informações no site da editora / Further information on publisher's website:
https://link.springer.com/chapter/10.1007/978-3-319-22174-8_18

DOI: 10.1007/978-3-319-22174-8_18

Direitos autorais / Publisher's copyright statement:
©2015 by Springer. All rights reserved.

DIRETORIA DE TRATAMENTO DA INFORMAÇÃO

Cidade Universitária Zeferino Vaz Barão Geraldo
CEP 13083-970 – Campinas SP
Fone: (19) 3521-6493
http://www.repositorio.unicamp.br

Fast Implementation of Curve25519 Using AVX2

Armando Faz-Hernández(✉) and Julio López

Institute of Computing, University of Campinas,
1251 Albert Einstein, Cidade Universitaria, Campinas, Brazil
{armfazh,jlopez}@ic.unicamp.br

Abstract. AVX2 is the newest instruction set on the Intel Haswell processor that provides simultaneous execution of operations over vectors of 256 bits. This work presents the advances on the applicability of AVX2 on the development of an efficient software implementation of the elliptic curve Diffie-Hellman protocol using the Curve25519 elliptic curve. Also, we will discuss some advantages that vector instructions offer as an alternative method to accelerate prime field and elliptic curve arithmetic. The performance of our implementation shows a slight improvement against the fastest state-of-the-art implementations.

Keywords: AVX2 · SIMD · Vector instructions · Elliptic Curve Cryptography · Prime Field Arithmetic · Curve25519 · Diffie-Hellman Protocol

1 Introduction

Nowadays, the use of Elliptic Curve Cryptography (ECC) schemes has been widely spread in secure communication protocols, such as the key agreement Elliptic Curve Diffie-Hellman (ECDH) protocol. In terms of performance, the critical operation in elliptic curve protocols is the computation of point multiplication. This operation can be accelerated by performing efficient computation of the underlying prime field arithmetic.

Recently, proposals of new elliptic curves defined over prime fields that accelerate the finite field arithmetic operations have appeared in [2,3,10]. Such proposals use pseudo-Mersenne primes ($p = 2^m - c$) which enable fast modular reduction. One of these proposals is based on the curve named *Curve25519*, which has been gained a lot of relevance due to their efficiency and secure implementation. The Curve25519 is a Montgomery elliptic curve defined over $\mathbb{F}_{2^{255}-19}$. On this curve, only the x-coordinate of a point P is required to compute the x-coordinate of the point multiplication kP, for any integer k.

For the past few years, processors have benefited from increasing support for vector instructions, which operate each instruction over a vector of data.

Armando Faz-Hernández and Julio López were partially supported by the Intel Labs University Research Office.

Julio López was partially supported by FAPESP, Projeto Temático grant number 2013/25.977-7.

© Springer International Publishing Switzerland 2015
K. Lauter and F. Rodríguez-Henríquez (Eds.): LatinCrypt 2015, LNCS 9230, pp. 329–345, 2015.
DOI: 10.1007/978-3-319-22174-8_18

The AVX2 instruction set extends the capabilities of the processor with 256-bit registers and instructions that are able to compute up to four simultaneous 64-bit operations. Thus, it is relevant to study how to benefit from vector instructions for the acceleration of ECC protocols. In this work, we exhibit implementation techniques using AVX2 instructions to compute the ECDH protocol based on Curve25519.

The rest of the document is presented as follows: in Sect. 2, the features of the AVX2 instruction set are described; in Sect. 3, we detail the prime field arithmetic for pseudo-Mersenne primes; in Sect. 4, Curve25519 is described along with the arithmetic of the Montgomery elliptic curve; in Sect. 5, we present the implementation techniques for the field $\mathbb{F}_{2^{255}-19}$ using AVX2 instructions; in Sect. 6, the performance results of our implementation are summarized; and finally in Sect. 7, we present the conclusions of this work.

2 The AVX2 Instruction Set

An interesting trend of micro-architecture design is the Single Instruction Multiple Data (SIMD) processing; in this setting, processors contain a special bank of vector registers and associated vector instructions, which are able to compute an operation on every element stored in the vector register. Since 1997, SIMD processing has been present on processors; first, starting with the MMX instruction set [12] which contains 64-bit vectors; and then followed by a number of Streaming SIMD Extensions (SSE) instruction sets [13] that extended the size of vector registers to 128 bits.

In 2011, the Advanced Vector eXtensions (AVX) instruction set was released. It extended the size of vector registers to 256 bits. However, most of the AVX instructions were focused on the acceleration of floating point arithmetic targeting applications for graphics and scientific computations, postponing the integer arithmetic instructions to later releases. Therefore, in 2013 Intel released the Haswell micro-architecture with the AVX2 instruction set [14], which contained plenty of new instructions not only to support integer arithmetic, but also to compute other versatile operations. For the purpose of this work, we detail the most relevant AVX2 instructions used, which will be referred by a mnemonic described in Table 4 (in Appendix A):

- Logic. The Xor and And instructions were extended to operate over every bit in a 256-bit register. The Align instruction concatenates simultaneously the lower (and higher) 128-bit parts from two 256-bit registers into a 256-bit temporary result, then shifts the result right by a multiple of 8 bits, and stores the lower (and higher) 128 bits in a destination register.
- Integer addition/subtraction (Add/Sub). AVX2 extended the integer addition and subtraction instructions from SSE and AVX to 256-bit vectors, thus enabling the computation of four simultaneous 64-bit operations. On AVX2 both addition and subtraction are unable to handle input carry and borrow.
- Integer multiplication (Mul). AVX2 is able to compute four products of 32×32 bits, storing four 64-bit results on a 256-bit vector register.

- Variable shifts. Former instruction sets were able to compute logical shifts SHL/SHR using the same fixed (resolved at compile time) shift displacement for every word stored in the vector register. Now, AVX2 added the new SHLV/SHRV variable shift instructions; thus, the displacement used for each word can be determined at run time. This feature adds more flexibility for the implementation of asymmetric operations over vector registers.
- Combination. The BLEND instruction fills the content of a vector register with the words from two different register sources chosen through a binary selection mask register; such mask can be defined either at compile or run time. The UNPCK instruction sets a register with the interleaved words of two registers.
- Permutation. The PERM, BCAST and PERM128 instructions move the words stored in a 256-bit vector register using a permutation pattern that can be specified either at compile or at run time.

In terms of performance, it is worth to say that the 4× speedup factor expected for 64-bit operations using AVX2 can be attained only for some instructions; in practice, factors like the execution latency of the instruction, the number of execution units available and the throughput of every instruction reduce the acceleration.

3 Prime Field Arithmetic Using Pseudo-Mersenne Primes

This section describes the techniques used for the efficient computation of the prime field arithmetic using a pseudo-Mersenne prime modulus. First, the representation of elements in a prime field is detailed; we then show how to perform each prime field operation under such a representation.

3.1 Representation of Prime Field Elements

Given an integer n (e.g. the size of machine registers), a commonly used approach to represent a field element $a \in \mathbb{F}_p$ is using a *multiprecision representation*:

$$A(n) = \sum_{i=0}^{s-1} u_i 2^{in}, \tag{1}$$

such that $a \equiv A(n) \bmod p$ for $n \in \mathbb{Z}^+$, $0 \leq u_i < 2^n$ and $s = \lceil \frac{m}{n} \rceil$. Using this representation an element is stored using s words of n bits. Multiprecision representation has been widely used in several multiprecision and cryptographic libraries [1,17,23].

However, one of the disadvantages of using a multiprecision representation on an n-bit architecture is that some arithmetic operations impose a sequential evaluation of integer operations; e.g. in the modular addition, the carry bits must be propagated from the least to the most significant coefficient, and this behavior limits the parallelism level of the computations. Since AVX2 has no support for

additions with carry, then a representation that minimizes the propagation of carry bits is required.

The *redundant representation* meets the criteria, because it relies on the selection of a real number $n' < n$; thus each word will have enough bits to store the carry bits produced by several modular additions. Thus, a field element $a \in \mathbb{F}_p$ in this representation is denoted by the tuple of coefficients $\mathbf{A} = \{a_{s'-1}, \ldots, a_0\}$ of the following number:

$$A(n') = \sum_{i=0}^{s'-1} a_i 2^{\lceil in' \rceil}, \tag{2}$$

where $a \equiv A(n') \bmod p$ for $n' \in \mathbb{R}$ and $s' = \left\lceil \frac{m}{n'} \right\rceil$. The fact that n' is a non-integer number produces that every coefficient a_i has an asymmetric amount of bits $\beta_i = \lceil n'(i+1) \rceil - \lceil n'i \rceil$ for $i \in [0, s')$.

The redundant representation introduces a significant improvement in the parallel execution of operations, a proof of such an acceleration was reported in [3], where the author proposed to use $n' = 25.5$ for speed up the elliptic curve arithmetic over $\mathbb{F}_{2^{255}-19}$.

3.2 Prime Field Operations

In order to compute prime field operations, the operands must be converted from binary to redundant representation and, at the end of the whole computation, the result must be converted back to binary.

Addition/Subtraction. Given two tuples $\mathbf{A}$ and $\mathbf{B}$, the operation $\mathbf{R} = \mathbf{A} \pm \mathbf{B}$ can be computed by performing the addition/subtraction coefficient-wise, e.g. $r_i = a_i \pm b_i$ for $i \in [0, s')$. Notice that these operations are totally independent and admit a parallel processing provided that no overflow occurs.

Multiplication. The computation of a prime field multiplication is usually processed in two parts: the integer multiplication and then the modular reduction; however as a pseudo-Mersenne prime ($p = 2^m - c$) is used to define the finite field then both operations can be computed in the same step. Therefore, given $\mathbf{A}$ and $\mathbf{B}$, the tuple $\mathbf{R} = \mathbf{A} \times \mathbf{B}$ is computed in the following manner:

$$r_i = \sum_{j=0}^{s'-1} (2^{\eta_{j,t}} \delta_{j,t}) a_j b_t \quad \text{for } i \in [0, s') \text{ and } t = i - j \bmod s', \tag{3}$$

where the terms $\delta_{x,y}$ and $\eta_{x,y}$ are constants defined as follows:

$$\delta_{x,y} = \begin{cases} c & \text{if } x + y \geq s', \\ 1 & \text{otherwise.} \end{cases} \tag{4}$$

$$\eta_{x,y} = \lceil xn' \rceil + \lceil yn' \rceil - \lceil (x + y \bmod s')n' \rceil \bmod m. \tag{5}$$

Since $\forall i \in [0, s')\ \beta_i \leq \lceil n' \rceil$ is true, this implies that the products in r_i on Eq. 3 will not overflow the $2n$-bit boundary for some $n' < n$. Additionally, whenever

$\delta_{x,y} \neq 1$ denotes those products that were moved around to the corresponding power of two because of modular reduction; and $\eta_{x,y} \neq 1$ indicates that some products must be adjusted as a consequence of n' not being an integer.

Squaring. Following the same idea for multiplication; in the squaring some products appear twice and then can be replaced with multiplications by 2, as denoted by term $\nu_{x,y} = 2$ if $x \neq y$, otherwise $\nu_{x,y} = 1$. Given a tuple $\mathbf{A}$, the square $\mathbf{R} = \mathbf{A}^2$ is computed as:

$$r_i = \sum_{j=\lceil \frac{i}{2} \rceil}^{\left\lfloor \frac{1}{2}(s'+i) \right\rfloor} (2^{\eta_{j,t}} \delta_{j,t} \nu_{j,t}) a_j a_t \quad \text{for } i \in [0, s'),\ t = i - j \bmod s', \tag{6}$$

Coefficient Reduction. Every time an addition, a subtraction, a multiplication or a squaring is computed, the result fits on s' words of n bits, so it is possible to continue processing more additions and subtractions. However, if the result of the operation is the input of a multiplication or of a squaring, then a coefficient reduction must be processed.

The coefficient reduction over a tuple $\mathbf{A}$ is an operation that ensures that every coefficient a_i verifies the following condition: $|a_i| \leq \beta_i + 1$, where β_i was defined in the previous section. This operation keeps the size of coefficients under a safe range to process another modular operation without overflowing registers.

Given a tuple $\mathbf{A}$, every coefficient is splitted into three parts, namely $a_i = h_i \parallel m_i \parallel l_i$, where $|l_i| = \beta_i$, $|m_i| = \beta_{i+1 \bmod s'}$, $|h_i| = n - |l_i| - |m_i|$ for $i \in [0, s')$, thus the coefficient reduction is computed as follows: $a'_0 = l_0 + t_0$, $a'_1 = l_1 + m_0 + t_1$, and $a'_i = l_i + m_{i-1} + h_{i-2}$ for $i \in [2, s')$, where the terms t_0 and t_1 are computed using the following equation: $t_1 2^{\beta_0} + t_0 = c \cdot (h_{s'-1} 2^{\beta_0} + m_{s'-1} + h_{s'-2})$.

Multiplicative Inverse. In order to compute the multiplicative inverse of an element $a \in \mathbb{F}_p^*$, the following identity is used: $a^{-1} \equiv a^{p-2} \pmod{p}$; part of this exponentiation can be calculated using an addition chain as shown by Itoh-Tsujii in [18]. Let $x, y \in \mathbb{Z}^+$ and $x \leq y$, define the term $\alpha_x = a^{2^x - 1}$ and the relation $\alpha_x \to \alpha_y$ as $\alpha_y = (\alpha_x)^{2^{y-x}} \alpha_{y-x}$. In [3] was given an addition chain for $\mathbb{F}_{2^{255}-19}$, starting with $\alpha_5 \to \alpha_{10} \to \alpha_{20} \to \alpha_{40} \to \alpha_{50} \to \alpha_{100} \to \alpha_{200} \to \alpha_{250}$, the multiplicative inverse is obtained as $a^{-1} = a^{2^{255}-21} = (\alpha_{250})^{2^5} a^{11}$ using 11 multiplications, 254 squarings and 265 coefficient reductions.

4 Elliptic Curve Diffie-Hellman on Curve25519

4.1 Safe Elliptic Curves

Around 2000, the National Institute of Standards and Technology (NIST) standardized a set of elliptic curves and associated parameters of finite fields to provide elliptic curve cryptography schemes for different security levels [20]. The selected elliptic curves were defined over both binary and prime fields. For the case of prime fields, prime numbers were selected as Generalized Mersenne

Table 1. Recent proposals of elliptic curves for three different security levels.

Security Level	Elliptic Curve	Prime Number (p)
128	NIST-P256	$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
	Curve25519	$2^{255} - 19$
	Curve1174	$2^{251} - 9$
192	NIST-P384	$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
	M-383	$2^{383} - 187$
	E-382	$2^{382} - 105$
256	NIST-P521	$2^{521} - 1$
	M-511	$2^{511} - 187$
	E-521	$2^{521} - 1$

primes, defined by Solinas in [22]; these primes have the property of allowing faster modular reduction compared to random selected primes.

Recently, new proposals have appeared for different elliptic curves which associate a different construction of prime modulus, such as [2,3,10]. The proposals have in common the use of pseudo-Mersenne primes ($p = 2^m - c$), with m being close to twice the targeted security level, and c as small as possible for the acceleration of prime field operations.

Nowadays, the study of the prime field implementation not only impacts on the efficiency of the cryptographic protocols but also on its security. An implementation of prime field arithmetic could cause leakage of secret information when is not implemented properly. Recently, Bernstein and Lange started the project called *SafeCurves* [7] with the aim to ensure elliptic curve cryptography security through the design of simple and secure implementations. The SafeCurves project evaluates the fulfillment of some security criteria over several elliptic curves. Table 1 lists some of the recent proposals of elliptic curves, notice that prime numbers selected are simpler than those from NIST's recommendation.

4.2 Arithmetic of Curve25519

Focussing on the 128-bit security level, the elliptic curve named Curve25519 has attracted some attention due to its efficient and secure implementation. For example, it has been proposed for inclusion in the DNS protocol (DNSCurve project [5]); additionally, the OpenSSH library has chosen Diffie-Hellman over Curve25519 as the default key-exchange protocol [21].

Curve25519 was proposed by Bernstein [3] for the acceleration of the elliptic curve Diffie-Hellman protocol targeting 128-bit security level. This curve is defined over the prime field $\mathbb{F}_{2^{255}-19}$ and has the following form:

$$\text{Curve25519}: \quad y^2 = x^3 + \hat{a}_2 x^2 + x, \qquad \hat{a}_2 = 486662. \tag{7}$$

This curve belongs to the family of Montgomery elliptic curves, which were used in [19] to accelerate the elliptic curve method for factoring (ECM).

Algorithm 1. Ladder Step Algorithm Tuned for SIMD Processing

Input: $X_{P-Q}, Z_{P-Q}, X_P, Z_P, X_Q, Z_Q \in \mathbb{F}_p$ and the coefficient $\hat{a}_2$ from Eq. (7).
Output: $X_{2P}, Z_{2P}, X_{P+Q}, Z_{P+Q} \in \mathbb{F}_p$.

1: $A \leftarrow X_P + Z_P$	$C \leftarrow X_Q + Z_Q$	[add]
2: $B \leftarrow X_P - Z_P$	$D \leftarrow X_Q - Z_Q$	[sub]
3: $DA \leftarrow A \times D$	$CB \leftarrow C \times B$	[mul]
4: $t_1 \leftarrow DA + CB$	$t_0 \leftarrow DA - CB$	[add/sub]
5: $t_1 \leftarrow t_1^2$	$t_0 \leftarrow t_0^2$	[sqr]
6: $X_{P+Q} \leftarrow t_1 \times Z_{P-Q}$	$Z_{P+Q} \leftarrow t_0 \times X_{P-Q}$	[mul]
7: $A' \leftarrow A^2$	$B' \leftarrow B^2$	[sqr]
8: $A'x \leftarrow \frac{1}{4}(\hat{a}_2 + 2) \cdot A'$	$B'y \leftarrow \frac{1}{4}(\hat{a}_2 - 2) \cdot B'$	[mul-cst]
9: $E \leftarrow A' - B'$	$F \leftarrow A'x - B'y$	[sub]
10: $X_{2P} \leftarrow A' \times B'$	$Z_{2P} \leftarrow E \times F$	[mul]
11: **return** $X_{2P}, Z_{2P}, X_{P+Q}, Z_{P+Q}$.		

In the same paper, Montgomery devised an algorithm to efficiently compute the x-coordinate of kP using only the x-coordinate of the point P; the technique uses the projective representation of points on the curve.

The computation of point multiplication using Montgomery Ladder algorithm is shown in Algorithm 6 in Appendix B.3. For each bit of the scalar, the procedure updates the values of two points P and Q through the ladder step algorithm (Algorithm 1), which computes a point doubling of P and a differential point addition of P and Q. The results are conditionally stored in temporary registers depending on a bit of the scalar k; the conditional update must be protected to avoid leaking the bits of k using either arithmetic or logic operations. Finally, the affine version of the x-coordinate of Q is recovered.

5 The AVX2 Implementation

This section starts discussing some performance penalties encountered in AVX2, then we describe some ways of getting a better performance through parallel computations in the Montgomery ladder algorithm, and finally we will show the techniques used to implement the prime field $\mathbb{F}_{2^{255}-19}$ with AVX2 instructions.

5.1 Performance Challenges Using AVX2

Before proceeding to the implementation of prime field operations, we detail a relevant issue on the implementation of modular multiplication. Recall that the AVX2 instruction MUL is able to process four integer multiplications of 32×32 bits, so in order to compute some products of the modular multiplication the natural approach is to pack four consecutive products. However, the way that the products are packed is critical in terms of performance; as an illustrative example, we present two cases that compute four products required in the modular multiplication, under the assumption that there are four registers initialized with

the following values: $R_0 = [a_3, a_2, a_1, a_0]$, $R_1 = [b_3, b_2, b_1, b_0]$, $R_2 = [b_7, b_6, b_5, b_4]$ and $R_3 = [\square, \square, b_9, b_8]$.

Example 1. To calculate the vector $[a_0b_3, a_0b_2, a_0b_1, a_0b_0]$, only two instructions are required: first, we fill a register with a_0 using the BCAST instruction and then we apply the MUL instruction with the register R_1.

Example 2. Computing $[a_3b_0, a_3b_9, a_3b_8, a_3b_7]$ requires the following set of operations:

$$
\begin{array}{ll}
X \leftarrow \textsc{Bcast}(R_0) & [a_3, a_3, a_3, a_3] \\
Y \leftarrow \textsc{Blend}(R_1, R_2, 1100) & [b_7, b_6, b_1, b_0] \\
Y \leftarrow \textsc{Perm}(Y) & [b_0, b_6, b_1, b_7] \\
U \leftarrow \textsc{Perm}(R_3) & [\square, b_9, b_8, \square] \\
Y \leftarrow \textsc{Blend}(Y, U, 0110) & [b_0, b_9, b_8, b_7] \\
Z \leftarrow \textsc{Mul}(X, Y) & [a_3b_0, a_3b_9, a_3b_8, a_3b_7]
\end{array}
$$

In the second example, the computation takes 5 instructions just to place the operands in the right position to be multiplied, while in the first example it only takes 1 instruction. Products arranged as in the second example appear more frequently in the computation of the modular multiplication; although we could compute them using permutation instructions, the use of these instructions impacts negatively on the performance of the operations.

The high latency of permutation instructions is the result of the architectural design of Haswell micro-architecture. The previous instruction sets (SSE and AVX) operate with an execution network that computes vector instructions on 128-bit registers. On the other hand, Haswell contains an additional network of 128-bit registers to represent the higher part of a 256-bit register, so both networks compute in parallel most of the AVX2 instructions. Consequently, any data transfer between such networks will incur a performance penalty.

5.2 The SIMD Montgomery Ladder

Since an efficient implementation of the prime field will improve the elliptic curve arithmetic; so, we also focus on the flow of operations in the curve level. Analyzing the ladder step algorithm, we noticed that there are several opportunities to compute two prime field operations in parallel without dependency between the elements involved in the operation.

The general idea is simple: it is possible to compute two prime field operations by packing the operands in the lower and higher parts of a 256-bit vector register, thus the arithmetic operations will be computed on both parts at the same time. At this point, some natural questions are raised: why do we not use a 4-way version in the evaluation of the operations? The answer comes from the evaluation of Montgomery ladder step algorithm, which processes a nearly symmetrical computation over two sets of prime field elements; this does not restrict the use of, for example, a 4-way version applied to computations with four independent operations in other scenarios. A second natural question is: using 2-way

prime field operations, are the benefits brought by the use of vector instructions lost? Working in this scenario with 256-bit registers, each prime field operation still takes advantage from the use of 128-bit registers.

A parallel computation of the ladder step algorithm was suggested in [11]: one of the parallel units will compute the point doubling while the other unit will produce the differential point addition. Another interesting idea is scheduling operations in a SIMD fashion, which was demonstrated to be efficient on the implementation presented in [9]; such an implementation takes advantage of the use of the NEON instructions to compute two finite field operations independently.

In this work, we go further by exploiting the parallelism at two levels: first at high level, the SIMD execution of prime field operations; and at low level, the computations inside of the prime field operation can also benefit from SIMD execution. The right-hand side of Algorithm 1 lists the operations computed in each row; as one may notice, the same operation is applied to two different data sets exhibiting exactly the spirit behind the SIMD paradigm. The way that the ladder step algorithm was presented in Algorithm 1 gives an insight of the register allocation and of the scheduling of operations.

5.3 Implementation of $\mathbb{F}_{2^{255}-19}$

For the implementation of $\mathbb{F}_{2^{255}-19}$ the most efficient approach is to set $n' = 51$ on a 64-bit architecture and $n' = 25.5$ for a 32-bit architecture. As was mentioned before, Bernstein in [3] encouraged the use of $n' = 25.5$; such a implementation used the floating point registers to emulate integer arithmetic operations because in that scenario a double precision floating point register is able to store a 53-bit number without loss of information. In our scenario, we also choose $n' = 25.5$ as working on a 32-bit architecture, because the wider multiplier available in AVX2 is a 32-bit multiplier, even though other fundamental integer operations support 64-bit operands.

Summarizing the parameters described in Sect. 3.1, our implementation of $\mathbb{F}_{2^{255}-19}$ sets $n' = 25.5$, $s' = 10$, $n = 32$ for integer multiplication and $n = 64$ for integer, shift and logic operations.

Interleaving Tuples. As it was shown in the previous section, the ladder step function computes two field operations at each time. We denote by $\langle \mathbf{A}, \mathbf{B} \rangle$ the interleaved tuples $\mathbf{A}$ and $\mathbf{B}$ which represents five 256-bit registers, such that $\langle \mathbf{A}, \mathbf{B} \rangle_i = [a_{2i+1}, a_{2i}, b_{2i+1}, b_{2i}]$ for $i \in [0, 5)$. Thus, two coefficients from tuple $\mathbf{A}$ will be stored in the higher 128-bit register and two coefficients from tuple $\mathbf{B}$ will be stored in the lower 128-bit register.

Addition/Subtraction. The addition and subtraction operations require only five addition (ADD) or subtraction (SUB) instructions, respectively.

Multiplication. Algorithm 2 shows the computation of two interleaved tuples $\langle \mathbf{A}, \mathbf{C} \rangle$ and $\langle \mathbf{B}, \mathbf{D} \rangle$. Values on the right hand side show the content stored in the destination register. The lines 2 to 5 compute the multiplication by the $\eta_{x,y}$ term

using variable shift instructions. In the main loop (lines 6–18), a temporary register U contains the first and third words of $\langle \mathbf{A}, \mathbf{C} \rangle_i$ in the lower and higher 128-bit parts of the register, respectively; this task can be efficiently performed using a SHUF instruction. Once that U was computed, in the inner loop U will be multiplied by $\langle \mathbf{B}, \mathbf{D} \rangle_j$ and the result will be accumulated in Z_{i+j}. Analogously to U, the products resulting from the V register will be accumulated in Z_{i+j+1}. Register W contains some products that must be accumulated in either Z_i or Z_{i+5}, thereby a BLEND instruction masks the appropriate products to be added. Finally, the products for which $\delta_{x,y} = 19$ will be multiplied using shift instructions.

Algorithm 2. Instruction scheduling to compute a modular multiplication in $\mathbb{F}_{2^{255}-19}$ using AVX2 instructions.

Input: Two interleaved tuples $\langle \mathbf{A}, \mathbf{C} \rangle$ and $\langle \mathbf{B}, \mathbf{D} \rangle$.
Output: An interleaved tuple $\langle \mathbf{E}, \mathbf{F} \rangle$ such that $\mathbf{E} = \mathbf{A} \times \mathbf{B}$ and $\mathbf{F} = \mathbf{C} \times \mathbf{D}$.

1: $Z_i \leftarrow 0$ for $i \in [0, 10)$
2: **for** $i \leftarrow 0$ **to** 4 **do**
3: $\quad \langle \mathbf{B}', \mathbf{D}' \rangle_i \leftarrow \text{ALIGN}(\langle \mathbf{B}, \mathbf{D} \rangle_{i+1 \bmod 5}, \langle \mathbf{B}, \mathbf{D} \rangle_i)$ $[b_{2i+2}, b_{2i+1}, d_{2i+2}, d_{2i+1}]$
4: $\quad \langle \mathbf{B}', \mathbf{D}' \rangle_i \leftarrow \text{SHLV}(\langle \mathbf{B}', \mathbf{D}' \rangle_i, [0, 1, 0, 1])$ $[b_{2i+2}, 2b_{2i+1}, d_{2i+2}, 2d_{2i+1}]$
5: **end for**
6: **for** $i \leftarrow 0$ **to** 4 **do**
7: $\quad U \leftarrow \text{SHUF}(\langle \mathbf{A}, \mathbf{C} \rangle_i, 0)$ $[a_{2i}, a_{2i}, c_{2i}, c_{2i}]$
8: $\quad$ **for** $j \leftarrow 0$ **to** 4 **do**
9: $\quad\quad Z_{i+j} \leftarrow \text{ADD}(Z_{i+j}, \text{MUL}(U, \langle \mathbf{B}, \mathbf{D} \rangle_j))$ $[a_{2i}b_{j+1}, a_{2i}b_j, c_{2i}d_{j+1}, c_{2i}d_j]$
10: $\quad$ **end for**
11: $\quad V \leftarrow \text{SHUF}(\langle \mathbf{A}, \mathbf{C} \rangle_i, 1)$ $[a_{2i+1}, a_{2i+1}, c_{2i+1}, c_{2i+1}]$
12: $\quad$ **for** $j \leftarrow 0$ **to** 3 **do**
13: $\quad\quad Z_{i+j+1} \leftarrow \text{ADD}(Z_{i+j+1}, \text{MUL}(V, \langle \mathbf{B}', \mathbf{D}' \rangle_j))$
$[a_{2i+1}b_{2j+2}, 2a_{2i+1}b_{2j+1}, c_{2i+1}d_{2j+2}, 2c_{2i+1}d_{2j+1}]$
14: $\quad$ **end for**
15: $\quad W \leftarrow \text{MUL}(V, \langle \mathbf{B}', \mathbf{D}' \rangle_4)$ $[a_{2i+1}b_0, 2a_{2i+1}b_9, c_{2i+1}d_0, 2c_{2i+1}d_9]$
16: $\quad Z_i \leftarrow \text{ADD}(Z_i, \text{BLEND}(W, [0, 0, 0, 0], 0101))$ $[a_{2i+1}b_0, 0, c_{2i+1}d_0, 0]$
17: $\quad Z_{i+5} \leftarrow \text{ADD}(Z_{i+5}, \text{BLEND}(W, [0, 0, 0, 0], 1010))$ $[0, 2a_{2i+1}b_9, 0, 2c_{2i+1}d_9]$
18: **end for**
19: **for** $i \leftarrow 0$ **to** 4 **do**
20: $\quad 19Z_{i+5} \leftarrow \text{ADD}(\text{ADD}(\text{SHL}(Z_{i+5}, 4), \text{SHL}(Z_{i+5}, 1)), Z_{i+5})$
21: $\quad \langle \mathbf{E}, \mathbf{F} \rangle_i \leftarrow \text{ADD}(Z_i, 19Z_{i+5})$
22: **end for**
23: **return** $\langle \mathbf{E}, \mathbf{F} \rangle$

Squaring/Coefficient Reduction. The description of the instruction scheduling for these operations can be found in Appendices B.1 and B.2, respectively.

Conditional Swapping. The Montgomery point multiplication requires the conditional swapping of register values depending on the bits of the scalar; usually, the scalar represents a secret key, thereby this operation must be computed without branches and must run in constant time. In order to implement these requirements, the conditional swapping is computed using logic operations as shown in Algorithm 3.

Algorithm 3. Conditional Swapping.

Input: $b \in \{0, 1\}$ a conditional bit, X and Y two registers to be swapped.
Output: $X \leftarrow Y$ and $Y \leftarrow X$ if $b = 1$, otherwise remain unchanged.
1: $M \leftarrow \text{Bcast}(-b)$
2: $T \leftarrow \text{And}(\text{Xor}(X, Y), M)$
3: $X' \leftarrow \text{Xor}(X, T)$
4: $Y' \leftarrow \text{Xor}(Y, T)$
5: **return** X', Y'

6 Performance Results

Benchmarking was performed on a Haswell processor (Core i7-4770) at 3.4 GHz, where the Intel Turbo Boost and Intel Hyper Threading technologies were disabled. Our code was compiled using the GNU C Compiler v4.9.0 and timings were measured as the average time of 10^6 and 10^4 computations for prime field operations and point multiplication, respectively.

Prime Field Operations. Table 2 shows the performance of the field arithmetic operations using AVX2 instructions. The first row exhibits the clock cycles required to compute one single arithmetic operation over a tuple $\mathbf{A}$; the second row represents the clock cycles used to compute two simultaneous arithmetic operations over interleaved tuples $\langle \mathbf{A}, \mathbf{B} \rangle$; and the last row shows the speedup factor obtained by the 2-way against the single implementation.

The acceleration of the 2-way operations was achieved by minimizing the use of permutation instructions and working with interleaved tuples. Recently, an algorithm to compute a modular multiplication on the field $\mathbb{F}_{2^{521}-1}$ using a redundant representation was presented in [16]. This algorithm requires $\frac{1}{2}s'(s'+1)$ word multiplications and $2(s'^2 - 1)$ word additions. The paper also shows a formulation for the field $\mathbb{F}_{2^{255}-19}$; and following this idea, we implemented a 2-way multiplier whose performance was 117 clock cycles, and this result is 48 % slower than our schoolbook 2-way multiplier that takes 79 clock cycles. The main issue observed was an overhead produced by arranging the vectors to be multiplied, as permuting words between registers is costly.

Elliptic Curve Diffie-Hellman. In order to illustrate the performance of our software implementation of Elliptic Curve Diffie-Hellman protocol using Curve25519, we follow the guidelines presented in [4] to implement the following algorithms:

- **Key Generation.** Let G be the generator point of the Curve25519 where $x(G) = 9$, the key generation algorithm computes a public key $x(kG)$ given a secret key $k \in [0, 2^{256})$.
- **Shared Secret.** Given the x-coordinate of the public key, $x(P)$, and a secret key k, the shared secret algorithm computes the x-coordinate of kP.

Table 3 shows the timings obtained by our implementation and compares the performance against the state-of-the-art implementations. For the key generation

Table 2. Cost in clock cycles to compute one prime field operation using single implementation, and two prime field operations using the 2-way implementation.

	Addition Subtraction	Multiplication	Squaring	Coefficient Reduction	Inversion
Single (1 op)	4	57	47	26	16,500
2-way (2 ops)	8	79	57	33	21,000
Speedup Factor	1×	1.44×	1.64×	1.57×	1.57×

Table 3. Timings obtained for the computation of the Elliptic Curve Diffie-Hellman protocol. The entries represent 10^3 clock cycles. The timings in the rows with $^{(\alpha)}$ were measured on our Core i7-4770 processor and the rest of the entries were taken from the corresponding references.

Implementation	Processor	Key Generation	Shared Secret
NaCl [8]	Core i7-4770 $^{(\alpha)}$	261.0	261.0
amd64-51 [6]	Core i7-4770	172.6	163.2
amd64-51 [6]	Xeon E3-1275 V3	169.9	161.6
Our work	Core i7-4770 $^{(\alpha)}$	156.5	156.5

algorithm the implementations listed in Table 3 use the same routine to compute both key generation and shared secret.

As it can be seen from Table 3, the performance achieved in both the key generation and the shared secret computations brings a moderate speedup compared against the implementations reported in the eBACS website [6]. Notice that non-vector implementations found in the literature take advantage of the native 64-bit multiplier that takes 3 clock cycles; whereas, for AVX2, the same computation is performed using a 32-bit multiplier that takes 5 clock cycles. Additionally, the high latency of some instructions guides the optimization to use a more reduced set of instructions.

On a side note, it is to be noted that, in the key generation algorithm, since $x(G) = 9$, the modular multiplication on line 6 of Algorithm 1 can be replaced by only a few shift instructions. In our implementation, this gives a 13.5 % speedup, computing the key generation step in only 135.5×10^3 clock cycles.

7 Conclusions

Applying vector instructions to an implementation requires a careful knowledge of the target architecture, thus selecting the best scheduling of instructions is not a straightforward task because it demands a meticulous study of the instruction set and of the architectural capabilities. On the presence of architectural issues that limit the performance, we found a way to overcome some of them and

produced an efficient implementation as fast and secure as the best optimized implementations for 64-bit architectures.

Our main contribution is a fast implementation of the elliptic curve Diffie-Hellman protocol based on Curve25519 with a minor improvement over the state-of-the-art implementations. This performance was mainly achieved due to the efficient implementation of $\mathbb{F}_{2^{255}-19}$ using AVX2 vector instructions.

In this work, we expose the versatility of the AVX2 instructions, where the SIMD processing was applied at two levels: at the higher level, we showed how to schedule arithmetic operations in the Montgomery ladder step algorithm to be computed in parallel; and at the lower level, the computation of prime field operations also benefited from vector instructions.

We remark that the algorithms used to implement prime field arithmetic using vector instructions can also be extended for other prime fields that use pseudo-Mersenne primes. The applicability of these techniques to other elliptic curve models is left as a future work. Also, it would be interesting to know how the upcoming architectures will impact on the performance of AVX2 instructions. In particular, the new Intel Skylake micro-architecture that has support for 512-bit registers should promote a high applicability of the results of this work.

Acknowledgments. The authors would like to thank the anonymous reviewers for their helpful suggestions and comments. Additionally, they would like to show their gratitude to Jérémie Detrey for his valuable comments on an earlier version of the manuscript.

Table 4. Latency and reciprocal throughput of some AVX2 instructions.

Type	Mnemonic	Assembler Instructions	Latency (cycles)	Reciprocal Throughput (cycles/op)
Arithmetic	Add/Sub	`VPADDQ/VPSUBQ`	1	0.5
	Mul	`VPMULDQ`	5	1
Logic	Shl/Shr	`VPSLLQ/VPSRLQ`	1	1
	Shlv/Shrv	`VPSLLVQ/VPSRLVQ`	2	2
	Align	`VPALIGNR`	1	1
	And/Xor	`VPAND/VPXOR`	1	0.33
Combination	Blend	`VPBLENDD`	1	0.33
		`VPBLENDVB`	2	2
	Shuf	`VSHUFPD`	1	1
	Unpck	`VPUNPCKHQDQ`	1	1
		`VPUNPCKLQDQ`	1	1
Permutation	Perm	`VPERMQ`	3	1
	Bcast	`VPBROADCASTQ`	5	0.5
	Perm128	`VPERM2I128`	3	1

A Relevant AVX2 Instructions

A list of the most relevant instructions used in this work is presented. For clarity, instructions were grouped according to their functionality. Table 4 shows in the second column a mnemonic used in this document; in the third column is described the specific assembler name of the instruction, and the last columns show the latency and the reciprocal throughput of every instruction, the entries were taken from the Agner Fog's measurements published in [15].

Algorithm 4. Instruction scheduling to compute a modular squaring in $\mathbb{F}_{2^{255}-19}$ using AVX2 instructions.

Input: An interleaved tuple $\langle \mathbf{A}, \mathbf{B} \rangle$.
Output: An interleaved tuple $\langle \mathbf{E}, \mathbf{F} \rangle$ such that $\mathbf{E} = \mathbf{A}^2$ and $\mathbf{F} = \mathbf{B}^2$.

1: **for** $i \leftarrow 0$ **to** 4 **do**
2: $U_{2i} \leftarrow \langle \mathbf{A}, \mathbf{B} \rangle_i$ $[a_{2i+1}, a_{2i}, b_{2i+1}, b_{2i}]$
3: $U_{2i+1} \leftarrow \textsc{Align}(\langle \mathbf{A}, \mathbf{B} \rangle_{i+1 \bmod 5}, \langle \mathbf{A}, \mathbf{B} \rangle_i)$ $[a_{2i+2}, a_{2i+1}, b_{2i+2}, b_{2i+1}]$
4: $U_{2i+1} \leftarrow \textsc{Shlv}(U_{2i+1}, [0, 1, 0, 1])$ $[a_{2i+2}, 2a_{2i+1}, b_{2i+2}, 2b_{2i+1}]$
5: $V_{2i} \leftarrow \textsc{Shuf}(\langle \mathbf{A}, \mathbf{B} \rangle_i, 0)$ $[a_{2i}, a_{2i}, b_{2i}, b_{2i}]$
6: $V_{2i+1} \leftarrow \textsc{Shuf}(\langle \mathbf{A}, \mathbf{B} \rangle_i, 1)$ $[a_{2i+1}, a_{2i+1}, b_{2i+1}, b_{2i+1}]$
7: **end for**
8: **for** $i \leftarrow 0$ **to** 4 **do**
9: $T \leftarrow \textsc{Mul}(U_i, V_i)$ $[a_{i+1}a_i, a_i a_i, b_{i+1}b_i, b_i b_i]$
10: $Z_i \leftarrow \textsc{Blend}(T, [0, 0, 0, 0], 1010)$ $[0, a_i a_i, 0, b_i b_i]$
11: $W \leftarrow \textsc{Blend}(T, [0, 0, 0, 0], 0101)$ $[a_{i+1}a_i, 0, b_{i+1}b_i, 0]$
12: **for** $j \leftarrow 1$ **to** i **do**
13: $t \leftarrow i - j \bmod 10$
14: $W \leftarrow \textsc{Add}(W, \textsc{Mul}(U_j, V_t))$ $[a_{j+1}a_t, a_j a_t, b_{j+1}b_t, b_j b_t]$
15: **end for**
16: $Z_i \leftarrow \textsc{Add}(Z_i, \textsc{Shl}(W, 1))$
17: $S \leftarrow \textsc{Mul}(U_{i+5}, V_{i+5})$ $[a_{i+6}a_{i+5}, a_{i+5}a_{i+5}, b_{i+6}b_{i+5}, b_{i+5}b_{i+5}]$
18: $Z_{i+5} \leftarrow \textsc{Blend}(S, [0, 0, 0, 0], 1010)$ $[0, a_{i+5}a_{i+5}, 0, b_{i+5}b_{i+5}]$
19: $X \leftarrow [0, 0, 0, 0]$
20: **for** $j \leftarrow i + 1$ **to** 4 **do**
21: $t \leftarrow i - j \bmod 10$
22: $X \leftarrow \textsc{Add}(X, \textsc{Mul}(U_j, V_t))$ $[a_{j+1}a_t, a_j a_t, b_{j+1}b_t, b_j b_t]$
23: **end for**
24: $Z_{i+5} \leftarrow \textsc{Add}(Z_{i+5}, \textsc{Shl}(X, 1))$
25: **end for**
26: **for** $i \leftarrow 0$ **to** 4 **do**
27: $19Z_{i+5} \leftarrow \textsc{Add}(\textsc{Add}(\textsc{Shl}(Z_{i+5}, 4), \textsc{Shl}(Z_{i+5}, 1)), Z_{i+5})$
28: $\langle \mathbf{E}, \mathbf{F} \rangle_i \leftarrow \textsc{Add}(Z_i, 19Z_{i+5})$
29: **end for**
30: **return** $\langle \mathbf{E}, \mathbf{F} \rangle$

B Algorithms

B.1 Implementation of Modular Squaring Using AVX2

To compute the modular squaring we follow a similar approach like in the case of modular multiplication. Algorithm 4 shows the scheduling of instructions used to compute the modular squaring of an interleaved tuple $\langle \mathbf{A}, \mathbf{B} \rangle$. The products $a_{x,y}$ such that $\nu_{x,y} = 2$ are computed in the inner loops (lines 12 to 15 and 20 to 23) and once that these products were accumulated, they are multiplied by 2 using shift instructions. At the end, the lines from 26 to 29 compute the modular reduction.

B.2 Implementation of Coefficient Reduction Using AVX2

The coefficient reduction is processed coefficient-wise. We split each coefficient into three parts $a_i = h_i \parallel m_i \parallel l_i$ and compute the process described in Sect. 3.2. Simultaneously, each m_i (medium coefficient) is added to the correspondent l_{i+1} (low coefficient) and to the h_{i-1} (high coefficient). For those coefficients that need to be reduced modulo p, we compute the multiplication by c using just shift instructions. After the coefficient reduction is processed, the size of each coefficient in the updated tuple will have at most $\beta_i + 1$ bits.

Algorithm 5. Instruction scheduling for computing a coefficient reduction in $\mathbb{F}_{2^{255}-19}$ using AVX2 instructions.

Input: An interleaved tuple $\langle \mathbf{A}, \mathbf{B} \rangle$.
Output: An updated interleaved tuple $\langle \mathbf{A}, \mathbf{B} \rangle$ such that $|a_i| \leq \beta_i + 1$ and $|b_i| \leq \beta_i + 1$ for $i \in [0, 10)$.

1: **for** $i \leftarrow 0$ **to** 4 **do**
2: $\quad L_i \leftarrow \textsc{And}(\langle \mathbf{A}, \mathbf{B} \rangle_i, [2^{\beta_{2i+1}} - 1, 2^{\beta_{2i}} - 1, 2^{\beta_{2i+1}} - 1, 2^{\beta_{2i}} - 1])$
3: $\quad M_i \leftarrow \textsc{Srlv}(\langle \mathbf{A}, \mathbf{B} \rangle_i, [\beta_{2i+1}, \beta_{2i}, \beta_{2i+1}, \beta_{2i}])$
4: $\quad M_i \leftarrow \textsc{And}(M_i, [2^{\beta_{2i+2}} - 1, 2^{\beta_{2i+1}} - 1, 2^{\beta_{2i+2}} - 1, 2^{\beta_{2i+1}} - 1])$
5: $\quad H_i \leftarrow \textsc{Srlv}(\langle \mathbf{A}, \mathbf{B} \rangle_i, [\beta_{2i+1} + \beta_{2i+2}, \beta_{2i} + \beta_{2i+1}, \beta_{2i+1} + \beta_{2i+2}, \beta_{2i} + \beta_{2i+1}])$
6: **end for**
7: **for** $i \leftarrow 0$ **to** 4 **do**
8: $\quad M'_i \leftarrow \textsc{Align}(M_i, M_{i-1 \bmod 5})$
9: **end for**
10: $H_4 \leftarrow \textsc{Shrv}(\langle \mathbf{A}, \mathbf{B} \rangle_8, [\beta_8 + \beta_9, \beta_8, \beta_8 + \beta_9, \beta_8])$
11: $U \leftarrow \textsc{Add}(H_4, \textsc{Shr}(H_4, 64))$
12: $19U \leftarrow \textsc{Add}(\textsc{Add}(\textsc{Shr}(U, 4), \textsc{Shr}(U, 1)), U)$
13: $T \leftarrow \textsc{And}(19U, [0, 2^{\beta_0} - 1, 0, 2^{\beta_0} - 1])$
14: $S \leftarrow \textsc{Shr}(19U, [0, \beta_0, 0, \beta_0])$
15: $H_4 \leftarrow \textsc{Upck}(T, S)$
16: **for** $i \leftarrow 0$ **to** 4 **do**
17: $\quad \langle \mathbf{A}, \mathbf{B} \rangle_i \leftarrow \textsc{Add}(\textsc{Add}(L_i, M'_i), H_{i-1 \bmod 5})$
18: **end for**
19: **return** $\langle \mathbf{A}, \mathbf{B} \rangle$

Algorithm 6. Point Multiplication using Montgomery Ladder.

Input: $k \in [0, 2^t)$ and $x(P) \in \mathbb{F}_p$, be the x-coordinate of an elliptic curve point P.
Output: $x(Q)$, the x-coordinate of $Q = kP$.

```
Let k = (0, k_{t-1}, ..., k_0)_2
X_{P-Q} ← x(P)
X_P ← x(P), Z_P ← 1
X_Q ← 1, Z_Q ← 0
for i ← t − 1 to 0 do
    b ← k_i ⊕ k_{i+1}
    X_Q, X_P ← CondSwap(b, X_Q, X_P)
    Z_Q, Z_P ← CondSwap(b, Z_Q, Z_P)
    X_Q, Z_Q, X_P, Z_P ← Ladder(X_{P-Q}, X_Q, Z_Q, X_P, Z_P)
end for
return x(Q) ← X_Q(Z_Q)^{-1}
```

B.3 Point Multiplication Using Montgomery Ladder

Algorithm 6 shows the computation of the Montgomery point multiplication to calculate the x-coordinate of kP given the x-coordinate of P and an integer scalar k. This algorithm also requires the ladder step presented in Algorithm 1.

For its use in the computation of the elliptic curve Diffie-Hellman protocol using the Curve25519, the document [4] describes an encoding for the secret key when is given as a string of bytes. Then, the description of Algorithm 6 assumes that the secret key was already encoded.

References

1. Aranha, D.F., Gouvêa, C.P.L.: RELIC is an Efficient LIbrary for Cryptography. http://code.google.com/p/relic-toolkit/
2. Aranha, D.F., Barreto, P.S.L.M., Pereira, G.C.C.F., Ricardini, J.E.: A note on high-security general-purpose elliptic curves. Cryptology ePrint Archive, Report 2013/647 (2013). http://eprint.iacr.org/
3. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006)
4. Bernstein, D.J.: Cryptography in NaCl, March 2009. http://cr.yp.to/highspeed/naclcrypto-20090310.pdf
5. Bernstein, D.J.: DNSCurve: usable security for DNS, June 2009. http://dnscurve.org
6. Bernstein, D.J., Lange, T.: eBACS: ECRYPT benchmarking of cryptographic systems, March 2015. Accessed on 20 March 2015 http://bench.cr.yp.to/supercop.html
7. Bernstein, D.J., Lange, T.: SafeCurves: choosing safe curves for elliptic-curve cryptography (2015). Accessed 20 March 2015 http://safecurves.cr.yp.to
8. Bernstein, D.J., Lange, T., Schwabe, P.: NaCl: Networking and Cryptography library, October 2013. http://nacl.cr.yp.to/

9. Bernstein, D.J., Schwabe, P.: NEON Crypto. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 320–339. Springer, Heidelberg (2012). http://dx.doi.org/10.1007/978-3-642-33027-8_19
10. Bos, J.W., Costello, C., Longa, P., Naehrig, M.: Selecting Elliptic Curves for Cryptography: An Efficiency and Security Analysis. Cryptology ePrint Archive, Report 2014/130 (2014). http://eprint.iacr.org/
11. Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., Vercauteren, F.: Handbook of Elliptic and Hyperelliptic Curve Cryptography, (2nd edn). Chapman & Hall/CRC (2012)
12. Corporation, I.: Intel Pentium processor with MMX technology documentation, January 2008. http://www.intel.com/design/archives/Processors/mmx/
13. Corporation, I.: Define SSE2, SSE3 and SSE4, January 2009. http://www.intel.com/support/processors/sb/CS-030123.htm
14. Corporation, I.: Intel Advanced Vector Extensions Programming Reference, June 2011. https://software.intel.com/sites/default/files/m/f/7/c/36945
15. Fog, A.: Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, December 2014
16. Granger, R., Scott, M.: Faster ECC over $\mathbb{F}_{2^{521}-1}$. Cryptology ePrint Archive, Report 2014/852 (2014). http://eprint.iacr.org/
17. Granlund, T., the GMP development team: GNU MP: The GNU Multiple Precision Arithmetic Library, (5.0.5 edn) (2012). http://gmplib.org/
18. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. Inf. Comput. **78**(3), 171–177 (1988). http://dx.doi.org/10.1016/0890-5401(88)90024–7
19. Montgomery, P.L.: Speeding the pollard and elliptic curve methods of factorization. Math. Comput. **48**(177), 243–264 (1987). http://dx.doi.org/10.2307/2007888
20. National Institute of Standards and Technology: Digital Signature Standard (DSS). FIPS Publication 186, may 1994. http://www.bibsonomy.org/bibtex/2a98c67565fa98cc7c90d7d622c1ad252/dret
21. Shell, O.S.: OpenSSH, January 2014. http://www.openssh.com/txt/release-6.5
22. Solinas, J.A.: Generalized Mersenne Numbers. Technical report,Center of Applied Cryptographic Research (CACR) (1999)
23. The OpenSSL Project: OpenSSL: The Open Source toolkit for SSL/TLS, April 2003. www.openssl.org