Fast FPGA Implementations of Diffie-Hellman on the Kummer Surface of a Genus-2 Curve

Philipp Koppermann[1], Fabrizio De Santis[2],
Johann Heyszl[1], and Georg Sigl[1,2]

[1] Fraunhofer Institute for Applied and Integrated Security, Munich, Germany
{philipp.koppermann,johann.heyszl,georg.sigl}@aisec.fraunhofer.de
[2] Technische Universität München, Munich, Germany
{desantis,sigl}@tum.de

Abstract. We present the first hardware implementations of Diffie-Hellman key exchange based on the Kummer surface of Gaudry and Schost's genus-2 curve targeting a 128-bit security level. We describe a single-core architecture for low-latency applications and a multi-core architecture for high-throughput applications. Synthesized on a Xilinx Zynq-7020 FPGA, our architectures perform a key exchange with lower latency and higher throughput than any other reported implementation using prime-field elliptic curves at the same security level. Our single-core architecture performs a scalar multiplication in 82 microseconds while our multi-core architecture achieves a throughput of 91,226 scalar multiplications per second. When compared to similar implementations of Microsoft's Four$\mathbb{Q}$ on the same FPGA, this translates to an improvement of 48% in latency and 40% in throughput for the single-core and multi-core architecture, respectively. Both our designs exhibit constant-time execution to thwart timing attacks, use the Montgomery ladder for improved resistance against SPA, and support a countermeasure against fault attacks.

Keywords: Diffie-Hellman key exchange, hyperelliptic curve cryptography, Kummer surface, FPGA, Zynq, low-latency, high-throughput, fault countermeasure.

1 Introduction

In 1989, Koblitz [15] first mentioned the application of hyperelliptic curves in cryptography. The Jacobian, which is associated to a genus-2 curve, enables a group structure that can be used for cryptographic algorithms such as Diffie-Hellman (DH) key exchange and digital signatures. Unfortunately, group operations on the Jacobian have higher complexity than those on elliptic curves (genus-1 curves). However, the Kummer surface of the Jacobian can be used to decrease the number of operations which are required for a (pseudo)-multiplication [12]. The Kummer surface is a 2-to-1 point mapping and can be compared to the x-coordinate-only representation of elliptic curves [1]. Table 1 shows the number of field operations for a point addition and a point doubling operation used

Table 1. Number of field operations for point addition and point doubling: multiplication (**M**), squaring (**S**), constant multiplication $\mathbf{M_c}$, addition (**A**), and subtraction (**Z**).

Genus	Reference	Field Size	M	S	$\mathbf{M_c}$	A	Z
1	Curve25519 [9]	255-bit	5	4	1	4	4
2	Kummer [18]	127-bit	7	12	12	16	16

in DH key exchange for a genus-1 Montgomery curve and a Kummer surface associated to a genus-2 curve. It can be noted that the genus-2 curve requires 1.4-times more multiplications, 3-times more squarings, and 4-times more additions and subtractions than the genus-1 curve. However, the Kummer surface based pseudo-multiplication operates on finite fields of half the size than those of elliptic curves while supporting the same security level. This reduced field size can lead to performance benefits in hardware, which is crucial for time critical applications.

In 2006, Bernstein [2] showed in a cost analysis for software that a genus-2 based implementation is potentially 1.5-times faster than a comparable elliptic curve based implementation. At that time, however, a secure Kummer surface of a genus-2 curve was not found yet. Since genus-2 point counting is computationally expensive, it took further six years until Gaudry and Schost [13] presented a twist secure Kummer surface targeting a 128-bit security level. Using this Kummer surface, Bernstein proved his earlier cost analysis [2] and presented a high-speed DH implementation on high-end CPUs that set new speed records [1]. These speed records were only surpassed by the Four$\mathbb{Q}$ elliptic curve implementation of Costello and Longa [6], which exploits a four-dimensional Gallant-Lambert-Vanstone decomposition to minimize the total number of group operations. Finally, Renes et al. [18] published implementation results of DH key-exchange on the Kummer surface of Gaudry and Schost's genus-2 curve [13] for different microcontrollers reporting notable performance results. So far, investigations of DH key exchange on the Kummer surface of genus-2 curves were confined to software implementations [1, 18] while no attention was paid to hardware implementations. In this work, we show that the Kummer surface of Gaudry and Schost's genus-2 curve can be used to perform very fast DH key-exchanges also in hardware.

Contribution We present the first FPGA implementations of Diffie-Hellman using the Kummer surface of a genus-2 hyperelliptic curve and show its competitiveness compared to elliptic curve based implementations. Following previous high-speed genus-2 implementations in software [1, 18], we use the Kummer surface [3, 4, 12] of Gaudry and Schost's genus-2 curve [13]. Synthesized on a Zynq-7020, our single-core architecture is about 1.91-times faster than the Four$\mathbb{Q}$ implementation [14], which is the so far fastest prime-field curve scalar multiplication on the same FPGA. In terms of throughput, our multi-core design shows

a factor-1.41 improvement compared to the Four$\mathbb{Q}$ implementation and a factor-2.82 improvement compared to the high-throughput Curve25519 implementation [20]. The main design decisions that allowed our results are summarized below.

Interleaving two scalar multiplications: Due to the serial nature of the considered ladder, multiple hardware modules operate below full capacity. However, this allows for a second scalar multiplication to be efficiently interleaved by carefully scheduling the required field operations. The obtained instruction schedule leaves the number of cycles unaltered while effectively doubling the throughput. Note that this interleaved scalar multiplication can also be used as a countermeasure against fault attacks by performing both scalar multiplications on the same input point and check the results for equivalence.

Efficient representation of constant values: For improved performance, we instantiate a dedicated circuit for multiplying field elements with 12-bit constants in each ladder step. Compared to a conventional modular multiplication, the constant modular multiplier requires only 4 clock cycles instead of 7. Some constants, however, are negative; the naive approach would be to convert them to positive elements of the prime field and then use the modular multiplier for multiplication. In order to avoid the increased memory requirements and decreased performance of this naive approach, we neglect the sign when storing the constants and include the conditional negation logic inside the constant modular multiplier.

High-speed modular multiplier: The performance of the scalar multiplication is strongly correlated with the performance of the modular multiplier. We take the multiplier presented in [10, 16], which is explicitly optimized for Mersenne prime fields, and modify it by applying non-standard tiling technique [19] to further improve its performance. In this way, we additionally reduce the number of required DSP blocks by 10%.

Organization The paper is organized as follows. In Sect. 2, we describe the basics of DH key exchange using a genus-2 curve, describe Gaudry and Schost's hyperelliptic curve and its Kummer surface, and summarize the scalar multiplication on this Kummer surface using the Montgomery ladder. In Sect. 3, a description of the single-core and multi-core hardware architectures is provided. In Sect. 4 we present the performance analysis and compare our results with related work. Finally, we conclude and discuss the results in Sect. 5.

2 Diffie-Hellman Key Exchange Using Kummer Surfaces

For elliptic and hyperelliptic curve based cryptosystems, the main operation is the scalar multiplication $Q = sP$, where Q, P are two points on a curve and $s \in \mathbb{Z}$ is a scalar value. In case of elliptic curve cryptography, the two points are located on the elliptic curve $\mathcal{E}$ satisfying the curve equation $\mathcal{E}(\mathbb{F}_q)$, where $\mathbb{F}_q$

is a finite field. An abelian group is formed by all points on the elliptic curve together with the point at infinity under the addition law, which is obtained by the chord-and-tangent rule. A point can be multiplied with a scalar by using an algorithm such as the Montgomery ladder [17], which repetitively performs point addition and point doubling operations.

For a genus-2 hyperelliptic curve $\mathcal{C}$, a group structure can be formed with the corresponding Jacobian $\mathcal{J}_\mathcal{C}(\mathbb{F}_q)$. The Jacobian forms an abelian group and is defined as the quotient space $\mathcal{J}_\mathcal{C} \coloneqq Div^0_\mathcal{C}/Princ_\mathcal{C}$, where $Div^0_\mathcal{C}$ denotes the group of divisors of degree zero of curve $\mathcal{C}$ and $Princ_\mathcal{C}$ the group of principal divisors of curve $\mathcal{C}$. A detailed description of Jacobians of hyperelliptic curves can be found in [5]. Although a point can be multiplied on the Jacobian, it is computationally expensive making it non-competitive compared to elliptic curve based implementations. Instead, we make use of the Kummer surface $\mathcal{K}_\mathcal{C}$ that is associated with $\mathcal{J}_\mathcal{C}$ of the hyperelliptic curve $\mathcal{C}$. The Kummer surface is defined as the quotient space of the Jacobian by its involution, which we denote by $\mathcal{K}_\mathcal{C} \coloneqq \mathcal{J}_\mathcal{C}/\langle\pm 1\rangle$. Gaudry [12] showed that scalar multiplication on the Kummer surface can be computed faster than on the corresponding Jacobian. Even though the group structure is lost when points on $\mathcal{J}_\mathcal{C}$ are mapped to $\mathcal{K}_\mathcal{C}$, a pseudo-multiplication [4] can be defined. For DH key exchange pseudo-multiplication is sufficient, and thus we perform all computations on $\mathcal{K}_\mathcal{C}$.

Our implemented DH key exchange works the same as the one described by Renes et al. [18]; we also follow their notation throughout this paper. If a point P is on the Jacobian $\mathcal{J}_\mathcal{C}$, we denote its image on the Kummer surface by $\pm P$. Each point $\pm P$ is represented by a 4-tuple where each element is 127-bit wide which sums up to 508 bit in total. As described in [4, 18], we assume that the public key (respectively public generator) is represented by a 3-tuple in its wrapped 381-bit representation denoted by $\underline{\pm P}$. Renes et al. [18] showed that keeping the input points in their wrapped representation offers two advantages: first, it reduces the required amount of data that needs to be transmitted and second, it results in a speed-up for the ladder computation.

For an ephemeral key exchange, the scalar multiplication is performed twice: once for computing an entity's public key, where the public generator is the input point, and once for computing a shared secret, where the other entity's public-key is the input point.

Key exchange. Let $\underline{\pm P}$ be the public generator (respectively public key) in its wrapped representation and s be the 251-bit secret key. We then compute $Q \leftarrow \pm[s]P$ and derive the generated public key (respectively the shared secret) as $\underline{\pm Q}$.

The scalar multiplication is implemented by Algorithm 1 (scalar_mult) and uses three functions: unwrap computes the 4-tuple representation of the input point, mont_ladder multiplies the unwrapped input point by a scalar value using the Montgomery ladder [17], and wrap finally computes the 381-bit wrapped representation of the output point; all these functions are described in detail in Sect. 2.3. As these functions depend on various parameters of the Kummer

Algorithm 1 scalar_mult: unwrap input point to Montgomery ladder on $\mathcal{K}_\mathcal{C}$ followed by point wrapping. It is assumed that the public key (respectively public generator) is in 381-bit wrapped representation.

Input: $\left(s = \sum_{i=0}^{250} s_i 2^i\right) \in [0, 2^{251}), \underline{\pm P}$ for $\pm P$ in $\mathcal{K}_\mathcal{C}$.
Output: $\underline{\pm Q}$ for $\pm Q \leftarrow \pm[s]P$ in $\mathcal{K}_\mathcal{C}$.

1: $\pm P \leftarrow \mathsf{unwrap}\left(\underline{\pm P}\right)$ ▷ compute 4-tuple representation of $\pm P$
2: $\pm Q \leftarrow \mathsf{mont\_ladder}\left(s, \pm P, \underline{\pm P}\right)$
3: $\underline{\pm Q} \leftarrow \mathsf{wrap}\left(\pm Q\right)$ ▷ compute wrapped 381-bit representation of $\pm Q$
4: **return** $\underline{\pm Q}$

surface of Gaudry and Schost's genus-2 hyperelliptic curve [13], we first summarize the definition of this curve in Sect. 2.1 and describe the associated Kummer surface in Sect. 2.2. More details can be found in [2, 18].

2.1 Gaudry and Schost's Genus-2 Hyperelliptic Curve

The genus-2 hyperelliptic curve $\mathcal{C}$ of Gaudry and Schost [13] is defined over the prime field $\mathbb{F}_p$ with $p = 2^{127} - 1$. The Rosenhain model of the curve $\mathcal{C}$ can be written as follows:

$$\mathcal{C} : Y^2 \coloneqq X\left(X-1\right)\left(X-\lambda\right)\left(X-\mu\right)\left(X-\nu\right) ,$$

where Rosenhain invariants are defined as

$$\begin{aligned}
\lambda &\coloneqq ac/bd &&= \texttt{0x1555555555555555555555555555552}\,,\\
\mu &\coloneqq ce/df &&= \texttt{0x73E334FBB315130E05A505C31919A746}\,,\\
\nu &\coloneqq ae/bf &&= \texttt{0x552AB1B63BF799716B5806482D2D21F3}\,,
\end{aligned}$$

the squared theta constants are set to

$$\begin{aligned}
a = -11\,, \quad b = 22\,, \quad c = 19\,, \quad \text{and} \quad d = 3\,,\\
e/f = (1 + \sqrt{CD/AB})/(1 - \sqrt{CD/AB})\,,
\end{aligned}$$

and the dual theta constants are set to

$$\begin{aligned}
A &\coloneqq a + b + c + d = 33\,, &\qquad B &\coloneqq a + b - c - d = -11\,,\\
C &\coloneqq a - b + c - d = -17\,, &\qquad D &\coloneqq a - b - c + d = -49\,.
\end{aligned}$$

2.2 Kummer Surface

Similar to previous works, we use the *fast* Kummer surface $\mathcal{K}_\mathcal{C} \in \mathbb{P}^3$ of [3, 4, 12], which is defined as:

$$\mathcal{K}_\mathcal{C} : Exyzt = \begin{pmatrix} \left(x^2 + y^2 + z^2 + t^2\right) \\ -F\left(xt + yz\right) - G\left(xz + yt\right) - H\left(xy + zt\right) \end{pmatrix}^2 ,$$

Algorithm 2 unwrap: $(x/y, x/z, x/t) \mapsto (x : y : z : t)$ unwrap point to its 508-bit representation.

Input: $(x/y, x/z, x/t) \in \mathbb{F}_p^3$.
Output: $(x : y : z : t) \in \mathbb{P}^3$.
1: $(V_1, V_2, V_3) \leftarrow ((x/z)(x/t), (x/y)(x/t), (x/y)(x/z))$
2: $V_4 \leftarrow V_3(x/t)$
3: **return** $(V_4 : V_1 : V_2 : V_3)$

Algorithm 3 wrap: $(x : y : z : t) \mapsto (x/y, x/z, x/t)$ compute wrapped 381-bit representation.

Input: $(x : y : z : t) \in \mathbb{P}^3$.
Output: $(x/y, x/z, x/t) \in \mathbb{F}_p^3$.
1: $V_1 \leftarrow yz$
2: $V_2 \leftarrow x/(V_1 t)$ ▷ inversion
3: $V_3 \leftarrow V_2 t$
4: **return** $(V_3 z, V_3 y, V_1 V_2)$

where

$$F = \frac{a^2 - b^2 - c^2 + d^2}{ad - bc}, \quad G = \frac{a^2 - b^2 + c^2 - d^2}{ac - bd}, \quad H = \frac{a^2 + b^2 - c^2 - d^2}{ab - cd},$$

and $E = 4abcd\,(ABCD/((ad - bc)(ac - bd)(ab - cd)))^2$. For a point P in $\mathcal{J}_\mathcal{C}$, its image $\mathcal{K}_\mathcal{C}$ is denoted by

$$(x_P : y_P : z_P : t_P) = \pm P\,.$$

The identity point $\langle 1, 0\rangle$ of $\mathcal{J}_\mathcal{C}$ maps to

$$\pm 0_{\mathcal{J}_\mathcal{C}} = (a : b : c : d)\ .$$

2.3 Scalar Multiplication on the Kummer Surface

As described in Algorithm 1 (scalar_mult), we assume that the input and output points are in their wrapped representation. The wrapped representation of the point $\pm P = (x : y : z : t)$ in $\mathcal{K}_\mathcal{C}$ is composed of a 3-tuple and denoted by $\underline{\pm P} = (x/y, x/z, x/t)$. Algorithm 2 (unwrap) implements the point unwrapping, which consists of 4 multiplications in $\mathbb{F}_p$. The wrapping function is described in Algorithm 3 (wrap); it consists of a finite field inversion and 7 multiplications. As in [18], we define three operations in the projective space $\mathbb{P}^3$ to improve the readability of the Montgomery ladder. First, the multiplication $\mathcal{M}$ that multiplies the corresponding pairs of coordinates from two distinct points in $\mathbb{F}_p$:

$$\mathcal{M} : ((x_1 : y_1 : z_1 : t_1), (x_2 : y_2 : z_2 : t_2)) \mapsto (x_1 x_2 : y_1 y_2 : z_1 z_2 : t_1 t_2)\ .$$

Algorithm 4 mont_ladder: Montgomery ladder using combined differential double-and-add.

Input: $\left(s = \sum_{i=0}^{250} s_i 2^i\right) \in [0, 2^{251}), \left(\pm P, \underline{\pm P}\right) \in \mathcal{K}_{\mathcal{C}}^2$.
Output: $\pm Q = (x_Q : y_Q : z_Q : t_Q) \in \mathbb{P}^3$ for $\pm Q \leftarrow \pm[s]P$ in $\mathcal{K}_{\mathcal{C}}$.

$V_5 \leftarrow (a : b : c : d)$
$V_6 \leftarrow (x_P : y_P : z_P : t_P)$ ▷ representation of $\pm P$
$V_7 \leftarrow \left(\frac{1}{A} : \frac{1}{B} : \frac{1}{C} : \frac{1}{D}\right)$
$V_8 \leftarrow \left(\frac{1}{a} : \frac{1}{b} : \frac{1}{c} : \frac{1}{d}\right)$
$V_9 \leftarrow \left(1 : \frac{x_P}{y_P} : \frac{x_P}{z_P} : \frac{x_P}{t_P}\right)$ ▷ representation of $\underline{\pm P}$
for $i = 250$ down to 0 **do**
    $(V_1, V_2) \leftarrow \mathsf{cswap}\left(s_i \oplus s_{i+1}, (V_5, V_6)\right)$ ▷ $s_{251} = 0$
    $(V_1, V_2) \leftarrow \left(\mathcal{H}\left(V_1\right), \mathcal{H}\left(V_2\right)\right)$
    $(V_3, V_4) \leftarrow \left(\mathcal{S}\left(V_1\right), \mathcal{M}\left(V_1, V_2\right)\right)$
    $(V_5, V_6) \leftarrow \left(\mathcal{M}\left(V_3, V_7\right), \mathcal{M}\left(V_4, V_7\right)\right)$
    $(V_1, V_2) \leftarrow \left(\mathcal{H}\left(V_5\right), \mathcal{H}\left(V_6\right)\right)$
    $(V_3, V_4) \leftarrow \left(\mathcal{S}\left(V_1\right), \mathcal{S}\left(V_2\right)\right)$
    $(V_5, V_6) \leftarrow \left(\mathcal{M}\left(V_3, V_8\right), \mathcal{M}\left(V_4, V_9\right)\right)$
end for
$(V_1, V_2) \leftarrow \mathsf{cswap}\left(s_0, (V_5, V_6)\right)$
return $\pm Q = V_2$

Second, the special case where the two points are equal, i.e. squaring in $\mathbb{F}_p$ the corresponding pairs of coordinates:

$$\mathcal{S} : (x : y : z : t) \mapsto \left(x^2 : y^2 : z^2 : t^2\right) .$$

Third, the Hadamard transform $\mathcal{H} : (x : y : z : t) \mapsto (x_{\mathcal{H}} : y_{\mathcal{H}} : z_{\mathcal{H}} : t_{\mathcal{H}})$ with

$$x_{\mathcal{H}} = \overbrace{(x+y)}^{u} + \overbrace{(z+t)}^{v}, \qquad z_{\mathcal{H}} = \overbrace{(x-y)}^{r} + \overbrace{(z-t)}^{s}, \tag{1}$$

$$y_{\mathcal{H}} = (x+y) - (z+t), \qquad t_{\mathcal{H}} = (x-y) - (z-t). \tag{2}$$

Finally, Algorithm 4 (mont_ladder) describes the Montgomery ladder for the scalar multiplication on the Kummer surface of Gaudry and Schost's genus-2 curve. The constants that are stored in V_7 and V_8 are projectively derived from the squared theta constants (a, b, c, d) and the dual theta constants (A, B, C, D) respectively (see Sect. 2.1):

$$\left(\frac{1}{a} : \frac{1}{b} : \frac{1}{c} : \frac{1}{d}\right) = (114 : -57 : -66 : -418) ,$$

$$\left(\frac{1}{A} : \frac{1}{B} : \frac{1}{C} : \frac{1}{D}\right) = (-833 : 2499 : 1617 : 561) .$$

The Montgomery ladder consists of 251 ladder steps, each one performing a differential-addition and a differential-doubling operation. Each ladder step includes a conditional swap of two pairs of coordinates.

3 Hardware Architectures

The implementation of Algorithm 1 (scalar_mult) is the essential task of our hardware design. We present a single-core architecture for low-latency applications and a multi-core architecture for high-throughput applications. Our single-core architecture performs two scalar multiplications on the Kummer surface at a time by scheduling the field operations for point addition and point doubling such that it is possible to interleave a second scalar multiplication with no cycle penalty. The top-view architecture is illustrated in Fig. 1.

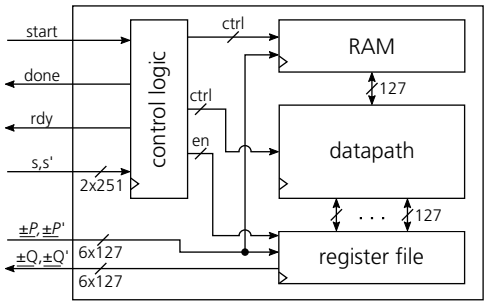

Fig. 1. Single-core architecture, which contains all control and datapath logic for computing Algorithm 1 (scalar_mult).

It takes two points in their wrapped representation as input, processes them, and returns two points in their wrapped representation as output. We logically divide our single-core design into three parts that are described in the next subsections: memory, datapath, and control logic. Further we describe a multi-core architecture that instantiates 4 independently operating cores and can perform up to 8 scalar multiplications with different keys and input points.

A Note on Fault Attacks The two interleaved scalar multiplications can be inherently used as a redundancy countermeasure to thwart fault attacks in our designs, i.e. by performing two interleaved scalar multiplications on the same points with the same key and then check the result for equivalence. This countermeasure can be applied to both our single- and multi-core architectures without applying any changes to the presented hardware designs.

3.1 Memory

The memory consists of a 16×127-bit register file and a 6×127-bit simple dual-port RAM. The register file is divided in four larger blocks, where each block is 4×127-bit wide. We follow the logical structure of Algorithm 4 (mont_ladder)

in which operations are performed on two points at a time (e.g. V_1, V_2 on line 8). We also use a simple dual-port RAM for storing the wrapped input point $\frac{x_p}{y_p}$, $\frac{x_p}{z_p}$, and $\frac{x_p}{t_p}$, which is accessed in read-only mode. Note that when no design constraints are set, the used synthesis tool instantiates distributed RAM instead of block RAM for storing this point. We found out that forcing the synthesis tool to use block RAM resulted in a $\approx 10\%$ decrease of the maximum clock frequency.

3.2 Datapath

The datapath including the register file is shown in Fig. 2. It implements the required field operations in $\mathbb{F}_p$. The register blocks R_i and R'_i for $i \in [1, 2]$ are

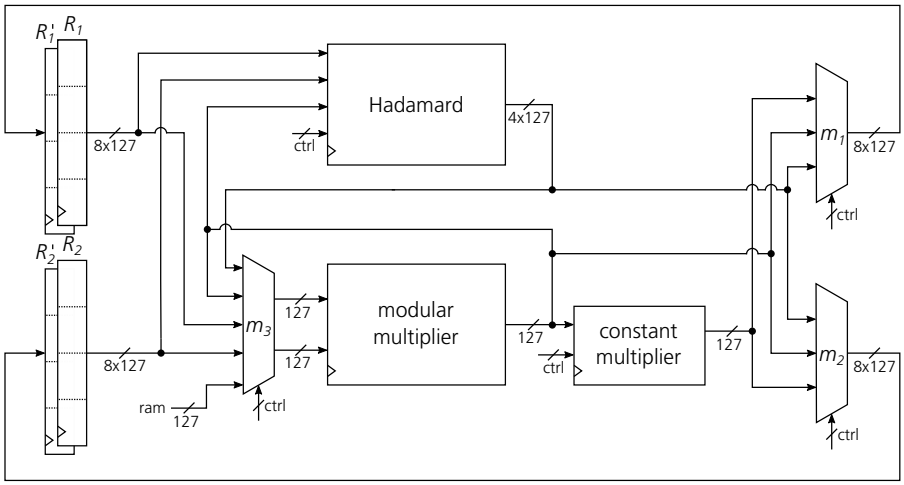

Fig. 2. Datapath including register file.

required for storing intermediate values of the first and the second scalar multiplication, respectively. The register blocks R_1 and R'_1 are initialized with the constants $V_5 = (a : b : c : d)$ whenever Algorithm 1 (scalar_mult) is started. The modular multiplier is preceded by the multiplexer m_3 that allows to perform field operations using various input sources. The output of the constant modular multiplier and the Hadamard module serve as fast forward input paths for the modular multiplier. These fast forward paths are required when data needs to be processed immediately without any further delay. Moreover, the modular multiplier can process 127-bit inputs that originate from the RAM and are required in each ladder step (e.g. multiplication by $\frac{x_p}{y_p}$). We can store each field operation output in the register blocks, i.e. R_i and R'_i, by accordingly selecting the signals with the multiplexers m_1 and m_2. Although large multiplexers

result in an increased area utilization, they allow greater flexibility in scheduling instructions which leads to higher overall performance. All select and enable signals in Fig. 2 are driven by the control logic (see Sect. 3.3).

Modular Multiplier We have implemented a modular multiplier that computes and accumulates its digit-products in full parallel. Combined with carefully placed pipeline stages, this parallel approach enables us to continuously fetch new input operands and return the result after 7 cycles including the reduction step. This property is not only beneficial for the performance, but also required in order to interleave a second scalar multiplication. Our implemented modular multiplier is used for both squaring and multiplication in $\mathbb{F}_p$. Fig. 3 shows the hardware architecture of our modular multiplier. After all digit-products

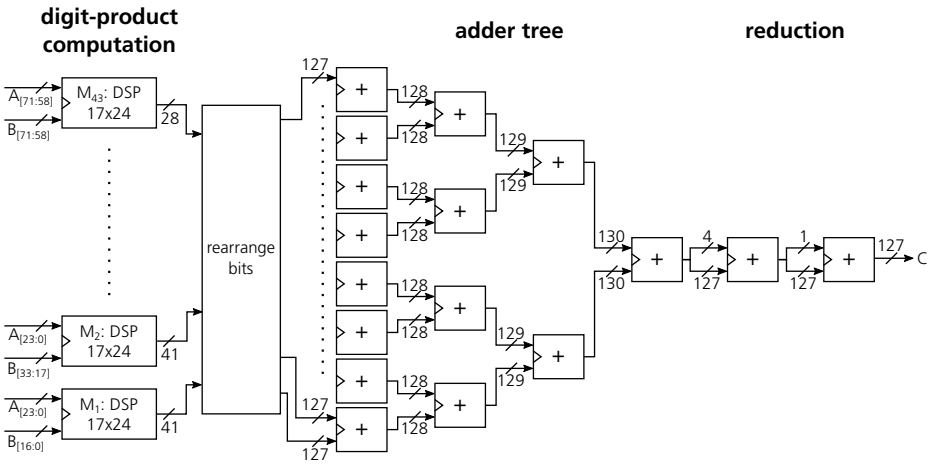

Fig. 3. Architecture of modular multiplier.

have been computed by the DSP blocks, they need to be summed up by an adder-tree. This adder-tree is commonly implemented in slower standard LUT logic and poses the bottleneck in most multiplier designs. A major problem is constituted by the large adder sizes that scale up to twice the operand width, i.e. 254-bit. Deriving a high-speed design is also complicated by the varying sizes of the adders in the adder tree, which can lead to inefficient pipelining. To overcome this problem, Koppermann et al. presented a technique in [16] for high-speed multiplication in Mersenne prime fields that reduces and equalizes the adder sizes. The main idea is to rearrange the digit-products on the bit-level while combining the multiplication with the fast reduction procedure proposed by Crandall [7] together.

In modern FPGAs, DSP blocks typically contain asymmetric multipliers, e.g. in case of the Zynq-7020 FPGA a 17×24-bit multiplier is contained in each

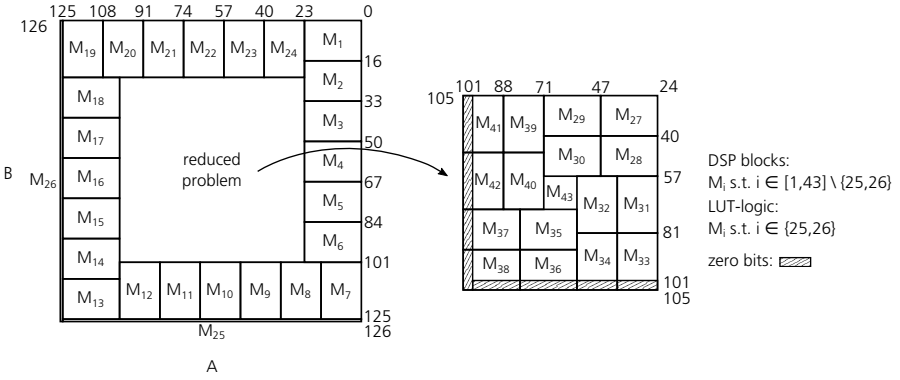

Fig. 4. Left: Non-standard tiling [19] for 127×127-bit multiplier. Right: Non-standard tiling for smaller 78×78-bit multiplier.

DSP block. In order to exploit these asymmetries to reduce the amount of DSP blocks used to perform large multiplications, different optimization strategies were proposed [8, 11, 21]. In particular, the authors of [8] showed that operand decomposition boils down to a tiling problem, where each tile represents the result of a smaller digit-product computation. Roy et al. [19] proposed the non-standard tiling algorithm as a solution to this tiling problem. They presented a formal procedure to compute this non-standard tiling for large multipliers with arbitrary operand sizes. For a 127×127-bit multiplier, Fig. 4 presents the implemented non-standard tiling [19]. The left side of Fig. 4 illustrates the initial tiling for the 127×127-bit multiplier. With this initial tiling, the problem of finding an efficient placement for a 127×127-bit multiplier is reduced to a 78×78-bit multiplier. Again, we perform non-standard tiling for the reduced problem which results in a smaller 14×14-bit multiplier M_{43}. The size of the tiles M_i where $i \in [1, 43] \setminus \{25, 26\}$ corresponds to the asymmetric multiplier widths and can consequently be implemented in a single DSP block. The two tiles M_{25} and M_{26}, however, correspond to a 126×1-bit multiplier and a 1×127-bit multiplier, respectively, both implemented in LUT logic. The horizontal side represents operand A and the vertical side represents operand B. Comparing non-standard-tiling with standard-tiling, only 41 DSP blocks are required instead of 64 [21].

Constant Modular Multiplier In order to speed up the Montgomery ladder, we instantiate a constant modular multiplier that multiplies one of the constants in $\left\{\frac{1}{a}, \frac{1}{b}, \frac{1}{c}, \frac{1}{d}, \frac{1}{A}, \frac{1}{B}, \frac{1}{C}, \frac{1}{D}\right\}$ with a variable 127-bit operand. The constant modular multiplier returns with a latency of 4 cycles, which is 3 cycles less than the generic modular multiplier, and is implemented using 6 DSP blocks only. The multiplication itself is pipelined and followed by two reduction steps including a conditional negation. The conditional negation is required for the multiplication with projectively negative constants, i.e. $\frac{1}{b}$, $\frac{1}{c}$, $\frac{1}{d}$, and $\frac{1}{A}$. For all other

constants, i.e. $\frac{1}{a}, \frac{1}{B}, \frac{1}{C}$, and $\frac{1}{D}$, the negation output is ignored. All constants are hard-decoded and then selected for multiplication via a select signal. Overall, 12 modular multiplications in each ladder step can be replaced by constant multiplications.

Hadamard Transform A core computation step in Algorithm 4 (mont_ladder) is the Hadamard transform. It is essentially composed of 4 modular additions and 4 modular subtractions, which we implemented using 2 modular adders and 2 modular subtractors. In order to parallelize the execution of independent operations, a modular adder is implemented using two addition circuits that are connected in series, each one having a clocked register output. The first adds two 127-bit wide operands and the second reduces the sum again by using Crandall's fast reduction [7]. Because a register is placed after each addition circuit, a result is obtained each cycle after an initial delay of 2 cycles. The modular subtraction circuit is implemented similarly; modular addition and modular subtraction are both implemented in LUT logic.

Two successive Hadamard transforms, i.e. $\mathcal{H}(V_1), \mathcal{H}(V_2)$, are computed at the beginning of each ladder step before any other computation can take place. Therefore, the modular adder and the modular subtractor circuits are connected with a multiplexer in a way that two Hadamard transforms are finished in successive clock cycles. Table 2 shows the scheduling for a Hadamard transform of two points, i.e. $V_1 = (x_1 : y_1 : z_1 : t_1)$ and $V_2 = (x_2 : y_2 : z_2 : t_2)$, plotted over cycles to compute Equation (1) and Equation (2) (see Sect. 2.3). The cycles plotted

Table 2. Instruction scheduling for two successive Hadarmard computations as in line 8 of Algorithm 4 (mont_ladder) using modular addition (**A**) and subtraction (**Z**).

	$\mathbf{A_1}$		$\mathbf{A_2}$		$\mathbf{Z_2}$		$\mathbf{Z_2}$	
cycle	1	3	1	3	1	3	1	3
1	u_1	-	v_1	-	r_1	-	s_1	-
2	u_2	-	v_2	-	r_2	-	s_2	-
3	$x_{\mathcal{H}_1}$	u_1	$z_{\mathcal{H}_1}$	v_1	$y_{\mathcal{H}_1}$	r_1	$t_{\mathcal{H}_1}$	s_1
4	$x_{\mathcal{H}_2}$	u_2	$z_{\mathcal{H}_2}$	v_2	$y_{\mathcal{H}_2}$	r_2	$t_{\mathcal{H}_2}$	s_2
5	-	$\mathbf{x_{\mathcal{H}_1}}$	-	$\mathbf{z_{\mathcal{H}_1}}$	-	$\mathbf{y_{\mathcal{H}_1}}$	-	$\mathbf{t_{\mathcal{H}_1}}$
6	-	$\mathbf{x_{\mathcal{H}_2}}$	-	$\mathbf{z_{\mathcal{H}_2}}$	-	$\mathbf{y_{\mathcal{H}_2}}$	-	$\mathbf{t_{\mathcal{H}_2}}$

under the corresponding component (e.g. modular adder $\mathbf{A_1}$) represent the processing stage. To give an example, u_1 in cycle 1 means that $u_1 = x_1 + y_1$ is in the first processing stage in the modular adder. In cycle 3, the computation of u_1 is finished and can be further processed by other modules. The transformed points $\mathcal{H}(V_1)$ and $\mathcal{H}(V_2)$ are returned in the 5th cycle and in the 6th cycle, respectively.

Table 3. Latency and throughput of field operations.

Operation	Latency (cycles)	Throughput (op/cycles)
Addition/subtraction in $\mathbb{F}_p$	2	1
Multiplication/squaring in $\mathbb{F}_p$	7	1
Constant multiplication in $\mathbb{F}_p$	4	1
Inversion in $\mathbb{F}_p$	952	1/476
Hadamard transform	4	1/2

A Note on Lazy Reduction To reduce the number of modular reductions and hence the number of required cycles, lazy reduction is a popular technique. In software, lazy reduction comes typically for free because field elements are often smaller than a multiple of the word size which results in unused bits at higher positions. In hardware, however, lazy reduction leads to increased memory requirements, larger multipliers, and a more complex control logic to distinguish between reduced and unreduced field elements when initiating a modular multiplication.

3.3 Control Logic

The control logic takes care of performing the necessary memory operations in the register file and RAM, and schedules the instructions required by Algorithm 1 (scalar_mult). The unwrapping and wrapping function, and the Montgomery ladder logically divide the control logic into separate control blocks. The control logic is implemented using a Finite State Machine (FSM). Inside the FSM multiple counters are used to track the processing status of arithmetic modules such as the modular multiplier. For an efficient instruction scheduling, the latency and throughput characteristics of the underlying functions such as modular multiplication and Hadamard transform are required. Table 3 shows the performance of the field operations in $\mathbb{F}_p$ and the Hadamard transform. The throughput denotes how often an instruction can be scheduled, e.g. a throughput of 1/2 (op/cycles) means 1 instruction can be scheduled in 2 cycles. Table 4 reports the latency of all high-level operations.

Montgomery Ladder Over 90 percent of all cycles are spent for the Montgomery ladder, and hence it is crucial to efficiently schedule field-level instructions. Table 5 shows the instruction scheduling for a Montgomery ladder step for two scalar multiplications. Instructions of the second scalar multiplication are complemented by a prime symbol, e.g. y_1'. Montgomery ladder calls 251 Montgomery ladder steps, each implementing a combined differential double-and-add which takes 41 cycles to run. All scheduled instructions denote the expected output, e.g. in cycle 5 the squaring y_3 is an abbreviation and stands for the computation of $y_3 = V_{3,y} = V_{1,y}V_{1,y}$ as described in line 9 of Algorithm 4 (mont_ladder).

Table 4. Latency of high-level functions.

Operation	**Latency (cycles)**
Unwrap	30
Combined differential double-and-add	41
Montgomery ladder	10,302
Wrap	998
Scalar multiplication	11,330

Table 5. Instruction scheduling for single ladder step as described in Algorithm 4 (mont_ladder) for the modular multiplier (**M**), the constant modular multiplier ($\mathbf{M_c}$), and the Hadamard transform module (**H**).

	M		**H**		$\mathbf{M_c}$			**M**		**H**		$\mathbf{M_c}$	
cycle	1	8	1	5	1	5	**cycle**	1	8	1	5	1	4
1	-	-	$\mathcal{H}_1$	-	-	-	28	z_3	z_3'	-	-	z_5'	y_6'
2	-	-	$\mathcal{H}_2$	-	-	-	29	t_3	t_3'	-	-	t_5'	z_6'
...	-	-	-	-	-	-	30	x_3	x_3'	-	-	x_5'	t_6'
5	y_3	-	-	$\mathcal{H}_1$	-	-	31	y_6	y_4	$\mathcal{H}_2'$	-	-	x_6'
6	y_4	-	-	$\mathcal{H}_2$	-	-	32	z_6	z_4	-	-	-	z_5'
7	z_4	-	-	-	-	-	33	t_6	t_4	-	-	-	t_5'
8	t_4	-	-	-	-	-	34	x_4	y_3	$\mathcal{H}_1'$	-	y_5	x_5'
9	x_4	-	-	-	-	-	35	y_4'	z_3	-	$\mathcal{H}_2'$	z_5	-
10	z_3	-	-	-	-	-	36	z_4'	t_3	-	-	t_5	-
11	t_3	-	-	-	-	-	37	t_4'	x_3	-	-	x_5	-
12	x_3	y_3	$\mathcal{H}_1'$	-	y_5	-	38	y_3'	$\mathbf{y_6}$	-	$\mathcal{H}_1'$	-	$\mathbf{y_5}$
13	-	y_4	$\mathcal{H}_2'$	-	y_6	-	39	z_3'	$\mathbf{z_6}$	-	-	-	$\mathbf{z_5}$
14	-	z_4	-	-	z_6	-	40	t_3'	$\mathbf{t_6}$	-	-	-	$\mathbf{t_5}$
15	-	t_4	-	-	t_6	-	41	x_3'	$\mathbf{x_4}$	-	-	-	$\mathbf{x_5}$
16	y_3'	x_4	-	$\mathcal{H}_1'$	x_6	y_5	1	y_6'	y_4'	-	-	-	-
17	y_4'	z_3	-	$\mathcal{H}_2'$	z_5	y_6	2	z_6'	z_4'	-	-	-	-
18	z_4'	t_3	-	-	t_5	z_6	3	t_6'	t_4'	-	-	-	-
19	t_4'	x_3	-	-	x_5	t_6	4	x_4'	y_3'	-	-	y_5'	-
20	x_4'	-	$\mathcal{H}_2$	-	-	x_6	5	-	z_3'	-	-	z_5'	-
21	z_3'	-	-	-	-	z_5	6	-	t_3'	-	-	t_5'	-
22	t_3'	-	-	-		t_5	7	-	x_3'	-	-	x_5'	-
23	x_3'	y_3'	$\mathcal{H}_1$	-	y_5'	x_5	8	-	$\mathbf{y_6}'$	-	-	-	$\mathbf{y_5}'$
24	y_4	y_4'	-	$\mathcal{H}_2$	y_6'	-	9	-	$\mathbf{z_6}'$	-	-	-	$\mathbf{z_5}'$
25	z_4	z_4'	-	-	z_6'	-	10	-	$\mathbf{t_6}'$	-	-	-	$\mathbf{t_5}'$
26	t_4	t_4'	-	-	t_6'	-	11	-	$\mathbf{x_4}'$	-	-	-	$\mathbf{x_5}'$
27	y_3	x_4'	-	$\mathcal{H}_1$	x_6'	y_5'	-	-	-	-	-	-	-

The conditional-swap function is implemented with no timing-penalty by simply swapping the arguments of the first two Hadamard transforms. Our control logic schedules modular multiplications and multiplications by constants in parallel for best performance results. Note that the constant multiplier uses the direct output of the modular multiplier.

Modular Inversion We use Fermat's little theorem to compute the multiplicative inverse x^{-1} of an integer $x \in \mathbb{F}_p \backslash \{0\}$. The finite field inversion is given by $x^{-1} \equiv x^{2^{127}-3}$.This exponentiation is computed with a sequence of 126 modular squarings and 10 modular multiplications as described by Renes et al. [18]. We implemented the modular inversion such that two elements of the prime field are inverted simultaneously by interleaving the field multiplication and squaring operations.

3.4 Multi-core Architecture

For multi-core architectures, the amount of cores which can be instantiated in parallel is strongly limited by the number of DSP blocks available on the target FPGA device. Our multi-core architecture implements 4 independently operating single-cores each featuring its own control logic. As a result, up to 8 scalar multiplications with different keys and input points can be computed. Instantiating multiple single-cores is a common concept and was similarly applied by Sasdrich and Güneysu [20] for Curve25519 and Järvinen et al. [14] for Four$\mathbb{Q}$. Sasdrich and Güneysu used a shared inversion module and Järvinen et al. used a shared control logic component. We also implemented a multi-core architecture with a shared control logic using a single shared key to reduce the area utilization. However, the LUT logic was only reduced by approximately 10% which is a rather small improvement compared to its limitations. In fact, this shared control logic architecture requires all scalar multiplications to be started in parallel as there is only one control logic for all cores.

4 Results and Analysis

We synthesized our single-core and multi-core architectures with Xilinx Vivado 2017.2 on a Xilinx Zynq-7020 FPGA (XC7Z020CLG484-3). All our results are obtained after place-and-route. Table 6 presents the area utilization including the maximum clock frequency for the single-core and multi-core architecture. Our single-core architecture requires 20% of the available slices and 22% of the available DSP blocks. We constrained our clock frequency to 138.7 MHz, which corresponds to a clock period of 7.21 ns. Two interleaved scalar multiplications require 11,330 cycles, and thus a session-key can be computed in 82 μs. Because we compute two scalar multiplication at a time, this translates to a throughput of 24,283 scalar multiplications per second. For our multi-core design we instantiate the maximum amount of 4 single-cores on the Zynq-7020 FPGA. Compared

Table 6. Device utilization and maximum clock frequency on Xilinx Zynq-7020 FPGA.

Component	Single-core @138.7 MHz	Multi-core @129.2 MHz	Available
LUTs	8,764 (16%)	35,015 (66%)	53,200
Registers	6,852 (6%)	27,300 (26%)	106,400
DSP48E1	49 (22%)	196 (89%)	220
Block RAM	0 (0%)	0 (0%)	140
Occupied slices	2,657 (20%)	10,554 (79%)	13,300

to our single-core design, we see a decrease in the maximum clock frequency; using Vivado tools, we can place-and-route our design with a clock frequency of 129.2 MHz which corresponds to a clock period of 7.74 ns. For the multi-core architecture with independently operating single-cores we report a throughput of 91,226 scalar multiplications per second.

Table 7 provides a comparison of our results with state-of-the-art scalar multiplication implementations on the same Zynq-7020 FPGA device all featuring a 128-bit security level. Namely, we compare our work with the Curve25519

Table 7. Comparison of single- and multi-core architectures with ECDH implementations featuring a 128-bit security level on a Zynq-7020.

Reference	Curve	Cores	Resources			Latency	T-put
			Slices	DSP	BRAM	(μs)	(op/s)
[20]	Curve25519	1	1,029	20	2	397	2,519
[14]	Four$\mathbb{Q}$ (Mont.)	1	565	16	7	310	3,222
[14]	Four$\mathbb{Q}$ (End.)	1	1,691	27	10	157	6,389
This work	**Kummer**	**1**	**2,657**	**49**	**0**	**82**	**24,283**
[20]	Curve25519	11	11,277	220	22	397	32,304
[14]	Four$\mathbb{Q}$ (End.)	11	5,697	187	110	170	64,730
This work	**Kummer**	**4**	**10,554**	**196**	**0**	**88**	**91,226**

implementation by Sasdrich and Güneysu [20] and the Four$\mathbb{Q}$ implementation by Järvinen et al. [14]. Comparing the latency of the single-core designs, our proposed implementation is 1.91-times faster than Four$\mathbb{Q}$ using endomorphisms, 3.78-times faster than Four$\mathbb{Q}$ using the Montgomery ladder, and 4.84-times faster than Curve25519. The improvement in latency is related to the increased slice and DSP block utilization. Yet, our implementation performs better than the fastest implementation so far (Four$\mathbb{Q}$ with End.) in the LUT-latency product (217,787 against 265,487) as well as the DSP-latency product (4,018 against 4,239). Our multi-core architecture with independently operating single-cores offers a throughput that is 1.41-times higher than Four$\mathbb{Q}$ and 2.82-times higher

than the Curve25519 implementation. In terms of latency, we also report the fastest scalar multiplication, i.e. our architecture is 1.93-times faster than Four$\mathbb{Q}$ and 4.51-times faster than Curve25519. Note that all reported multi-core designs use the maximum number of cores that can be successfully placed on the target device. However, only our multi-core design features fully independent single-cores, i.e. neither the inversion unit (e.g. Curve25519 implementation [20]) nor the scalar multiplication unit (e.g. Four$\mathbb{Q}$ implementation [14]) are shared. Also note that we make use of distributed RAM implemented by LUT logic for memory, which leaves a notable amount of BRAM available for other applications.

5 Conclusions

We presented the first hardware implementation results for a key exchange on the Kummer surface of Gaudry and Schost's genus-2 curve. Although a Kummer surface based key exchange has an increased number of field operations per ladder step when compared to elliptic curves, our presented architectures perform a scalar multiplication with lower latency and higher throughput than any other reported prime-field elliptic curve key exchange featuring a 128-bit security level on a Zynq-7020 FPGA. Our single-core architecture achieves a latency of 82 μs with a throughput of $24,283$ operations per second while the multi-core architecture with 4 independently operating single-cores achieves a latency of 88 μs with a throughput of $91,226$ operations per second. These results set new records for latency and throughput among state-of-the-art 128-bit secure key exchange implementations known so far, such as Curve25519 [20] and Four$\mathbb{Q}$ [14].

Acknowledgements

The authors acknowledge Abhijith Chikrapla Danappa and Zohaib Khan for their valuable effort in developing the initial prototypes of the presented implementations.

References

1. Bernstein, D.J., Chuengsatiansup, C., Lange, T., Schwabe, P.: Kummer Strikes Back: New DH Speed Records, pp. 317–337. Springer Berlin Heidelberg, Berlin, Heidelberg (2014), https://eprint.iacr.org/2014/134.pdf
2. Bernstein, D., Lange, T.: Elliptic vs. hyperelliptic, part 1. Talk at ECC p. 4 (2006), http://cr.yp.to/talks.html#2006.09.20
3. Chudnovsky, D., Chudnovsky, G.: Sequences of Numbers Generated by Addition in Formal Groups and New Primality and Factorization Tests. Adv. Appl. Math. 7(4), 385–434 (Dec 1986), http://dx.doi.org/10.1016/0196-8858(86)90023-0
4. Chung, P.N., Costello, C., Smith, B.: Fast, uniform scalar multiplication for genus 2 Jacobians with fast Kummers. Cryptology ePrint Archive, Report 2016/777 (2016), https://eprint.iacr.org/2016/777

5. Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., Vercauteren, F.: Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second Edition. Chapman & Hall/CRC, 2nd edn. (2012)
6. Costello, C., Longa, P.: Four$\mathbb{Q}$: Four-Dimensional Decompositions on a $\mathbb{Q}$-curve over the Mersenne Prime, pp. 214–235. Springer Berlin Heidelberg, Berlin, Heidelberg (2015), http://dx.doi.org/10.1007/978-3-662-48797-6_10
7. Crandall, R.: Method and apparatus for public key exchange in a cryptographic system (1992), US Patent 5,159,632
8. Dinechin, F.D., Pasca, B.: Large multipliers with fewer DSP blocks. In: 2009 International Conference on Field Programmable Logic and Applications. pp. 250–255 (Aug 2009)
9. Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C., Sánchez, A.H., Schwabe, P.: High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers (2015), http://eprint.iacr.org/2015/343
10. Fraunhofer Institute for Applied and Integrated Security: GitHub repository: modmul-mersenne. https://github.com/Fraunhofer-AISEC (2017)
11. Gao, S., Al-Khalili, D., Chabini, N., Langlois, P.: Asymmetric large size multipliers with optimised FPGA resource utilisation. IET Computers Digital Techniques 6(6), 372–383 (November 2012)
12. Gaudry, P.: Fast genus 2 arithmetic based on Theta functions. Journal of Mathematical Cryptology JMC 1(3), 243–265 (2007)
13. Gaudry, P., Schost, É.: Genus 2 point counting over prime fields. Journal of Symbolic Computation 47(4), 368 – 400 (2012), http://www.sciencedirect.com/science/article/pii/S0747717111001386
14. Järvinen, K., Miele, A., Azarderakhsh, R., Longa, P.: Four$\mathbb{Q}$ on FPGA: New Hardware Speed Records for Elliptic Curve Cryptography over Large Prime Characteristic Fields (2016), http://dx.doi.org/10.1007/978-3-662-53140-2_25
15. Koblitz, N.: Hyperelliptic cryptosystems. Journal of Cryptology 1(3), 139–150 (1989), http://dx.doi.org/10.1007/BF02252872
16. Koppermann, P., De Santis, F., Heyszl, J., Sigl, G.: Automatic generation of high-performance modular multipliers for arbitrary mersenne primes on FPGAs. In: 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 35–40 (May 2017)
17. Montgomery, P.L.: Speeding the pollard and elliptic curve methods of factorization. Mathematics of computation 48(177), 243–264 (1987)
18. Renes, J., Schwabe, P., Smith, B., Batina, L.: μKummer: efficient hyperelliptic signatures and key exchange on microcontrollers (2016), http://eprint.iacr.org/2016/366
19. Roy, D.B., Mukhopadhyay, D., Izumi, M., Takahashi, J.: Tile before multiplication: An efficient strategy to optimize DSP multiplier for accelerating prime field ECC for NIST curves. In: 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6 (June 2014)
20. Sasdrich, P., Güneysu, T.: Implementing Curve25519 for Side-Channel–Protected Elliptic Curve Cryptography. ACM Trans. Reconfigurable Technol. Syst. 9(1), 3:1–3:15 (Nov 2015), http://doi.acm.org/10.1145/2700834
21. Srinath, S., Compton, K.: Automatic Generation of High-performance Multipliers for FPGAs with Asymmetric Multiplier Blocks. In: Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays. pp. 51–58. FPGA '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1723112.1723123