Architecture level Optimizations for Kummer based HECC on FPGAs

Gabriel Gallin, Turku Ozlum Celik, Arnaud Tisserand

Architecture level Optimizations for Kummer based HECC on FPGAs

Gabriel GALLIN[1], Turku Ozlum CELIK[2], and
Arnaud TISSERAND[3][0000−0001−7042−3541]

[1] CNRS, IRISA UMR 6074, INRIA Centre Rennes - Bretagne Atlantique and University Rennes 1, Lannion, France
gabriel.gallin@irisa.fr
[2] IRMAR, University Rennes 1, Rennes, France
turku-ozlum.celik@univ-rennes1.fr
[3] CNRS, Lab-STICC UMR 6285 and University South Britany, Lorient, France
arnaud.tisserand@univ-ubs.fr

Abstract. On the basis of a software implementation of Kummer based HECC over $\mathbb{F}_p$ presented in 2016, we propose new hardware architectures. Our main objectives are: definition of architecture parameters (type, size and number of units for arithmetic operations, memory and internal communications); architecture style optimization to exploit internal parallelism. Several architectures have been designed and implemented on FPGAs for scalar multiplication acceleration in embedded systems. Our results show significant area reduction for similar computation time than best state of the art hardware implementations of curve based solutions.

Keywords: hyper-elliptic curve cryptography, hardware implementation, architecture exploration, embedded systems.

1 Introduction

Reducing the cost of asymmetric cryptography is a challenge for hardware implementation of embedded systems where silicon area is limited. Hyper-elliptic curve cryptography (HECC [6]) is considered to be an interesting solution compared to elliptic curve cryptography (ECC [12]). HECC requires smaller finite fields than ECC at similar security level. For instance, size of field elements is divided by two in genus-2 HECC solutions. But the number of field-level operations is larger in HECC per key/scalar bit. Then comparisons depend a lot on curve parameters, algorithm optimizations and implementation efforts.

HECC solutions based on Kummer surfaces (see [10] for details) demonstrate promising improvements for embedded software implementations. In 2016, Renes *et al.* presented in [24] a new Kummer-based HECC (KHECC) solution and its implementation on microcontrollers with 30 to 70 % clock cycles count reduction compared to the best similar curve based solutions at equivalent security level.

To the best of our knowledge, there is no hardware implementation of this recent KHECC solution. Below, we present hardware architectures for KHECC

adapted from [24] for scalar multiplication and their FPGA implementations. We study and evaluate the impact of various architecture parameters on the cost and performances: type, size and number of units (arithmetic, memory, internal communications); architecture topology; and exploitation of internal parallelism. We target embedded applications where the FPGA bitstream cannot be changed easily (trusting the configuration system at application level is complex) or prototyping for ASIC applications. Then to provide flexible circuits at software level, our solutions are designed for $\mathbb{F}_p$ with generic primes (where [24] only deals with $p = 2^{127} - 1$). Several architectures have been designed and implemented on different FPGAs for various parameters, and compared in terms of area and computation time.

Our paper outline is as follows. Section 2 recalls background on KHECC and introduces notations. Section 3 presents major elements of the work [24] used as a starting point and discusses required adaptations for hardware implementation. Section 4 quickly presents our units selected for the architectures and our tools used for design space exploration. Section 5 describes the proposed KHECC architectures and their implementation results on different FPGAs. Section 6 reports comparisons. Finally, Section 7 concludes the paper.

2 State-of-the-Art

HECC was introduced by Koblitz in [14] as a larger set of curves compared to ECC with a generalization of the class of groups obtained from the jacobians of hyper-elliptic curves. Subsequently, many HECC improvements have been proposed. See book [6] for a complete presentation on HECC and book [12] for ECC. Broadly speaking, field elements in HECC are smaller than in ECC for a similar security level (*e.g.* 128-bit HECC on genus-2 curves is equivalent to 256-bit ECC). This reduction should directly benefit to HECC since the width of field elements has a major impact on circuit area. But HECC requires more field operations to achieve operations at curve level such as point addition (`ADD`) and point doubling (`DBL`) for each scalar/key bit.

Many efforts have been made to reduce the cost of curve level operations in HECC. For genus-2, one can refer to Lange [18], Gaudry [10], Bos *et al.* [5] and Renes *et al.* [24] for instance. Table 1 reports a few costs for HECC and ECC solutions. There were many works on $\mathbb{F}_{2^n}$ solutions at low security levels (fields with 80–90 bits) in the past but very few on $\mathbb{F}_p$ at 128-bit security level until recently in software (our goal is hardware implementation).

KHECC solutions from [24] are based on a Kummer surface $\mathcal{K}_\mathcal{C}$ of an hyper-elliptic curve $\mathcal{C}$ defined over $\mathbb{F}_p$. Curve $\mathcal{C}$ is defined using constant parameters among which the "squared theta constants" (a, b, c, d) used during scalar multiplications. Points are represented by tuples of four n-bit coordinates in $\mathbb{F}_p$ where $\pm P = (x_P : y_P : z_P : t_P)$ is the projection of P from $\mathcal{C}$ on $\mathcal{K}_\mathcal{C}$.

In (H)ECC primitives such as signature or key exchange, the main operation is the *scalar multiplication* $[k]P_b$ of a base point P_b by a m-bit scalar or key k. In embedded systems, scalar multiplication must be protected against *side channel*

attacks (SCAs [20]). A popular protection against SCAs is the adaptation of *Montgomery ladder* (ML) algorithm [22]. ML is *constant time* (*i.e.* computation time of iterations does not depend on the key bit values) and *uniform* (*i.e.* the exact same schedule of the exact same field operations is executed at each iteration whatever the key bit values).

For KHECC hardware implementations (but also for more general HECC solutions), designers have to face several questions. How one should exploit the internal parallelism available at field level? Are few large and fast units more efficient than several parallel small and slow units? How to select parameters in a parallel architecture? Our work was related to those questions.

3 Hardware Adaptation of Renes *et al.* Solution

We based our work on the KHECC solution presented by Renes *et al.* at CHES 2016 [24] for software implementations of Diffie-Hellman key exchange and signature at 128-bit security level. Their solution optimizes the use of Kummer surface of hyper-elliptic curve described by Gaudry [10].

3.1 Analysis of Renes *et al.* Solution

In [24], the prime for $\mathbb{F}_p$ is $p = 2^{127} - 1$ due to fast modular reduction algorithms for Mersenne primes. The scalar size is $m = 256$ bits. ML algorithm starts with most significant key bits first. Each iteration computes a couple of points $(\pm V_1, \pm V_2)$ of $\mathcal{K}_\mathcal{C}$ from the result of the previous iteration using curve level operations `CSWAP` and `xDBLADD`. Using initial values $\pm V_1 = (a : b : c : d)$ and $\pm V_2 = (x_{P_b} : y_{P_b} : z_{P_b} : t_{P_b})$, the scalar multiplication computes $(\pm[k]P_b, \pm[k+1]P_b)$.

The core operation in ML iterations is the modified pseudo-addition `xDBLADD` *combined differential double-and-add* (see [24]). Given points $\pm V_1, \pm V_2$ on $\mathcal{K}_\mathcal{C}$, and base point $\pm P_b$, it computes $(\pm[2]V_1, \pm(V_1 + V_2)) = \texttt{xDBLADD}(\pm V_1, \pm V_2, \pm P_b)$. Based on the set of $\mathbb{F}_p$ operations described in Figure 1, `xDBLADD` has a *constant time* and *uniform* behavior.

The `CSWAP` operation consists in swapping the 2 input points (IN) of `xDBLADD` and the 2 resulting points (OUT) depending on the current key bit value (see Algo. 7 in [24]). It does not involve any computation but it impacts SCA aspects.

Renes *et al.* performed a smart selection of optimized curve parameters (from [10]) to determine constants with reduced size: 16 bits instead of 127.

solution & source	field width [bit]	`ADD`	`DBL`
$\mathbb{F}_p$ ECC [4]	ℓ_{ECC}	12M + 2S	7M + 3S
$\mathbb{F}_{2^n}$ HECC [18]	$\ell_{\text{HECC}} \approx 0.5\ell_{\text{ECC}}$	40M + 4S	38M + 6S
$\mathbb{F}_p$ KHECC [24]	$\ell_{\text{HECC}} \approx 0.5\ell_{\text{ECC}}$	19M + 12S	

Table 1. Cost per key bit of curve level operations in various (H)ECC solutions (`M` and `S` denote multiplication and square in the finite field).

Then they use a dedicated optimized function for modular multiplication by this type of constants in the software implementation.

Their implementations target low-cost microcontrollers: 8-bit AVR AT Mega and 32-bit ARM Cortex M0. They report significant improvements over the best known solutions. On Cortex M0, the clock cycles count is reduced by 27 % for key exchange and by 75 % for scalar multiplication in signature. On AT Mega, the corresponding reductions are respectively 32 % and 71 %.

3.2 Objectives and Constraints for our Hardware Accelerators

Unlike Renes *et al.* [24], in the present paper we only propose hardware acceleration for scalar multiplication since this is the main operation in terms of performance, energy consumption and security against SCAs (when the scalar is the private key). As is frequently the case in a complete embedded system, we assume that our hardware accelerator is coupled to a software implementation for high level primitives (which are out of scope of this paper).

In order to design flexible hardware accelerators and to report results in a general case, we target KHECC on generic prime fields. In [24] the selected prime $p = 2^{127} - 1$ leads to very cheap modular reduction but it is very specific (there is no Mersenne prime for slightly different security levels). Currently, we only deal with generic primes but we plan to derive versions for specific ones (*e.g.* pseudo-Mersenne) in the future. Field characteristic impacts the choice of curve parameters. We propose to use material presented in the work from Gaudry [10] to derive curves parameters and implementation constants.

One of our goal is to study hardware accelerators for scalar multiplication at architecture level. KHECC offers some internal parallelism as illustrated in Figure 1. Groups of 4 to 8 $\mathbb{F}_p$ operations can be easily performed at the same time with uniform and constant time schedules. In most of ECC solutions, fewer operations can be performed in parallel. We will evaluate the impact of this parallelism on the design of efficient accelerators with various trade-offs in terms of area and computation time (see questions at end of Section 2).

This paper is not dedicated to protection against physical attacks. But we target hardware accelerators where the execution of ML type of algorithms is actually constant time and uniform at low level. We will describe how we designed some units to achieve this objective (not yet evaluated using real attacks).

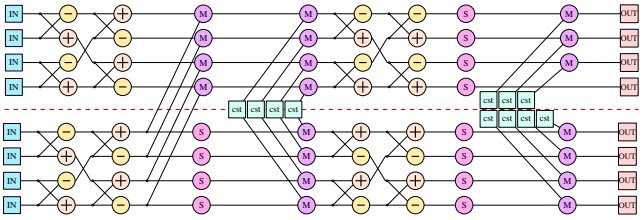

Fig. 1. $\mathbb{F}_p$ operations in `xDBLADD` from [24] (IN/OUT are $\pm V_1$, $\pm V_2$ coordinates).

In order to provide hardware accelerators that can be easily adapted to other application constraints and algorithms, we use a modular type of architectures based on independent units (for arithmetic operations and internal storage); a microcoded control; and an internal communication system based on multiplexors (between some units). This type of modular architectures allows us to easily explore many solutions in design space: type, size and number of units; size of internal communications; scheduling impact on performances and security. This choice also comes from the type of dedicated resources available in modern FPGAs: small embedded multipliers (DSP slices/blocks), small embedded memory (BRAM: block RAM), dedicated multiplexors in routing resources. Architecture modularity also helps the debug and validation process of the proposed solutions using a hierarchical method. First, we extensively and individually simulate in HDL and evaluate on FPGAs all our units. Second, we extensively simulate in HDL and evaluate on FPGAs the complete accelerator solutions.

4 Accelerator Units and Exploration Tools

Architecture modularity allows to explore a wide parameter space but it requires huge efforts in terms of implementation and debug. First, we decided to design, implement, validate, and optimize a small set of units for arithmetic operations, memory and internal communications. They are presented in Section 4.1. Second, based on this set of units, we designed specific tools for architecture level exploration and evaluation. These tools are quickly described in Section 4.2.

4.1 Accelerator Units and Resources

There are several types of resources in our hardware accelerators:

- arithmetic units for field level operations ($\mathbb{F}_p$ addition, subtraction and multiplication with generic prime p in this work);
- memory unit(s) for storing intermediate values ($\mathbb{F}_p$ elements for points coordinates), curve parameters and constants;
- a `CSWAP` unit in charge of scalar management and on-line schedule of operands depending on scalar/key bit values;
- an internal communication system for data transfers between the units;
- a control based on a microcode running the architecture.

Multiplier (`Mult`): Prime p is generic and can be programmed in our architecture. Then, there is no low-cost modular reduction. We use the *Montgomery modular multiplication* (MMM) proposed in [21]. Operands and product are represented in *Montgomery domain* (MD) $\left\{(x \times R) \bmod p, \forall x \in \mathbb{F}_p\right\}$ with $\gcd(R, p) = 1$ and $R > p$. We use MMM optimization methods from Orup [23] and Koc *et al.* [15] where internal high-radix computations reduce the amount of data dependencies and partial product/reduction steps are interleaved to speed up the multiplication with a slightly larger internal datapath.

In our target Xilinx FPGAs (see Section 5), there are embedded multipliers for 18×18 bits signed integers called DSP blocks/slices (in 2's complement representation). But for $\mathbb{F}_p$ computations, one can only use the 17 LSBs (all but the "sign" bit). Then our datapath width must be a multiple of 17. We evaluated that 34-bit word size for multiplication is interesting in our KHECC context (smaller size leads to slow multipliers, larger size requires too many DSP slices). For 34-bit words, operands and products are between 0 and $2p$ represented using *136 bits* in MD (136 is the closest multiple of 17 and 34 larger than 127). Then we can select $R > 4p$ for speed purpose (as many works in state of the art).

For improving the efficiency of our accelerators, we used the *hyper-threaded* multiplier version we proposed in [9] specially for KHECC. Hyper-threading hides the latency in DSP slices at high frequency (when all internal registers are activated). It computes 3 independent MMMs in parallel using 11 DSP slices (17×17), 2 BRAMs and a few slices in 79 clock cycles at 360 MHz on Virtex 5.

Due to MD, all field elements, parameters and constants are 128-bit values (contrary to [24] where shorter constants can be used). Using shorter constants in hardware requires a specific unit and then a more complex control and smaller overall frequency. Also for flexibility and frequency reasons, we only use generic $\mathbb{F}_p$ multiplier `Mult`. For the same reasons, we avoid dedicated square units (for generic p, MMM variants for square operation only lead to small improvements).

Similarly to state of the art solutions, final reduction from MD ($0 \leq x < 2p$) to "standard" $\mathbb{F}_p$ ($0 \leq x < p$) is performed after the scalar multiplication (there is no performance or security issue for this conversion).

Adder (`AddSub`): It performs both modular addition/subtraction ($(x \pm y) \bmod 2p$) in MD by setting a mode signal at operation start-up. Subtraction is implemented by adding operand x with the 2's complement of y operand ($\bar{y}$). The reduced sum in MD (resp. difference), is obtained from the parallel computations $r = x + y$ (resp. $r = x + \bar{y}$) and $r_p = r + \overline{2p}$ (resp. $r_p = r + 2p$). The output is r if $0 \leq r < 2p$, else r_p. The range of r is determined from the value of the output carry bit of $r + \overline{2p}$ for addition and of $x + \bar{y}$ for subtraction. We evaluated the impact of `AddSub` units for several word sizes: 34, 68 and 136 bits. The two large ones significantly reduce the overall frequency of the accelerator (due to longer carry propagations). It seems that our target FPGAs are optimized to handle word sizes around 32 bits but not for larger widths without costly pipeline schemes. To enforce short combinatorial paths and simplify the control, we set `AddSub` internal datapath width to $w_{\text{arith}} = 34$ bits as in `Mult`.

Memory: The accelerator uses internal memory(ies) for storing intermediate values ($\mathbb{F}_p$ elements of points coordinates), curve parameters and some constants (*e.g.* initial values $(a : b : c : d)$). To fit the internal width of arithmetic units ($w_{\text{arith}} = 34$ bits) and BRAM width in Xilinx FPGAs (configurable into $\{1, 2, 4, 9, 18, 36\}$ bits), we selected a memory configuration where words are multiple of 34 bits to simplify the control and avoid interfaces. Due to the large number of memory operations (read/write), we will show in Section 5 that a wider memory reduces the number of clock cycles per memory operation of full 136-bit values. We tested 3 configurations for the memory width (and internal

config.	w [bit]	s [word]	cycle(s) / mem. op.	BRAM(s)
`w34`	34	4	4	1
`w68`	68	2	2	2
`w136`	136	1	1	4

Table 2. Memory and internal communication width configurations.

communications see below) described in Table 2. The main parameter w (in bits) is the internal width of memory words (and communications). Related parameter s is the number of words required for storing a complete 136-bit value. Our BRAMs are configured into 512 lines of 36 bits words (2 unused bits per word). Less than 512 words are required for KHECC even in `w34` configuration. Table 2 also reports the clock cycles count of each memory operation and the memory area (in BRAMs). For security reason, this internal memory in restricted to the accelerator and cannot be accessed from outside (inputs and outputs are handled by a specific very small unit which is mute during scalar multiplications).

Internal communications: The units are interconnected through a specific internal communication system based on multiplexors (buses are not very efficient in target FPGAs and lead to high capacitances switching which can be a bad point for SCA protection). The communication system will be described in Section 5. In order to explore cost and performance trade-offs, we used configurations from Table 2 for the width in the internal communication system. Different widths for memory and communications requires a very costly control. Then w is shared for memory and communications. But for arithmetic units in this paper, we evaluated that $w_{\text{arith}} = 34$ is the best choice for our KHECC accelerators. For `w68` and `w136` configurations, small serial-parallel interfaces are added in the arithmetic units to handle the width difference with communications.

`CSWAP` unit: In algorithm 7 from [24] (called `crypto_scalarmult`), the `CSWAP` operation manages the scalar/key bits by swapping, or not, points $\pm V_1$ and $\pm V_2$ at the beginning and end of each ML iteration. We designed a dedicated unit for this purpose. It reads 2 $\mathbb{F}_p$ elements as inputs (corresponding to one coordinate of $\pm V_1$ and $\pm V_2$) and swaps them, or not, depending on the actual key bit value for the current iteration. At the last iteration, `CSWAP` triggers a "end of scalar multiplication" signal. In our `CSWAP` unit, there is no variable addresses or key management in the accelerator control for security reasons (instructions decoding does not depend on secret bits). For SCA protection, our `CSWAP` unit has been designed to ensure that there is always electrical activity in the pipelined unit, communication system and memory between successive clock cycles even if there is no swapping (read/write operations for one coordinate are interleaved with those of the other coordinates). Our `CSWAP` unit (pipelined with internal communications and memory) has a constant time and uniform behavior.

In the future, we will investigate the use of advanced scalar recoding schemes on the performances and security against SCAs.

Accelerator control: We defined a tiny ISA (instruction set architecture). Instructions, detailed in Table 3, are read from the code memory and decoded to

instruc.	description
`read`	transfer operands from memory to target unit and start computation
`write`	transfer result from target unit to memory
`wait`	wait for `immediate` clock cycles
`nop`	no operation (1 clock cycle)
`jump`	change program counter (PC) to `immediate` code address
`end`	trigger the end of the scalar multiplication

Table 3. Instructions set for our accelerators.

provide control signals to/from units, communication system and memory. The user program is stored into a small program memory (one BRAM). This type of control provides flexibility, avoids long synthesis and place&route processes (during modifications of user programs), and leads to fairly high frequencies on the target FPGAs when using pipelined BRAMs and decoding.

Instructions are 36-bit wide and our KHECC programs fit into one single BRAM (<512 instructions). Instructions contain: 4-bit opcode, 3-bit unit index, 2-bit operation mode, two 9-bit memory addresses and 9-bit immediate value.

We implemented hardware loops, using small finite state machines (FSMs), to handle s cycles during communication and memory operations and duration of `wait` instruction. Instruction decoding does not handle or depend on the scalar bit values for SCA protection (only the `CSWAP` unit handles secret bits).

Control resources include a few w-bit registers dedicated to external communications and initialization of memory parameters. They are very small and not involved during scalar multiplications, then we do not detail them here.

In the future, we plan to explore other types of control (*e.g.* distributed or FSM based solutions without microcode) for ASIC implementations.

4.2 Tools for Exploration and Evaluation at Architecture Level

Several parameters must be specified at design time for each architecture:

- type and number of units (`AddSub`, `Mult`, `CSWAP`, memory);
- width w for internal memory and communications;
- topology of the architecture.

All units have been fully described in synthesizable VHDL for FPGA implementation (with optimizations for DSP slices and BRAMs). They can also be tested and evaluated using *cycle accurate and bit accurate* (CABA) simulations. Then the time model at every clock cycle and the hardware cost of each unit are perfectly known (from implementation results).

Fully designing all possible architectures in VHDL is too time consuming. We decided to define and use a hierarchical and heterogeneous method to efficiently explore and validate numerous architectures.

Each architecture is described and simulated using a high-level model based on a CCABA (critical CABA) specification[4]. The *critical cycles* at architecture level are clock cycles where there are transitions in the control signals to/from the units and their inputs/outputs. For functional units, this corresponds to operands inputs, operation mode selection, start of computation, end of computation, and results outputs. The purely internal control signals inside the units are not modeled in CCABA (since their behavior is perfectly determined in the VHDL description and does not impact other parts of the accelerator).

A CCABA simulation tool has been developed in Python. Each unit is modeled in Python to specify: a) its mathematical behavior and b) its behavior at critical cycles based on the corresponding VHDL model (*e.g.* computation duration after start signal). For each unit, we need its complete VHDL and Python CCABA models (both manually written). Our tool allows fast simulations of complete scalar multiplications due to the hierarchical approach.

We also started the development of a tool to automatically schedule arithmetic and memory operations as well as internal communications. Currently, it uses a basic greedy algorithm to first feed the multipliers (due to their longer latency). We plan to improve it in the future. The schedule gives the total clock cycles count of a complete scalar multiplication.

We are able to quickly estimate the area and computation time of various architectures. The estimated area sums up the VHDL results for all units instantiated in the accelerator. The computation time is estimated by the total number of clock cycles multiplied by the slowest unit period. We approximate the impact of the control system (not yet designed in VHDL at this stage of the exploration) based on our experience.

During the exploration, we perform this type of estimation for each accelerator configuration to be evaluated. Then we select the most interesting solutions for full implementation in VHDL, final validation, and accurate comparisons (the corresponding results are presented in Section 5).

Once the accelerator has been fully implemented in VHDL, it is intensively tested using both VHDL simulations and executions on FPGA cards against reference values computed by SAGE mathematical software.

5 Proposed Architectures

We explored various configurations for parameters and architectures. We selected 4 architectures summarized in Table 4 and fully implemented them in VHDL on several FPGAs. The corresponding results are reported in sub-sections below. We began with a small and basic architecture A1 embedding the minimum number of units. Then we explored optimizations and more parallel architectures. Architecture A2 uses an optimization of `CSWAP` unit (V2). Architecture A3 embeds 2 operators for each arithmetic operation ($\pm$ and $\times$) to reach a higher parallelism

[4] Our CCABA model is inspired by Transaction Level Modeling (TLM) with full cycle accuracy for all control signals at the architecture level but not inside the units (when there is no input/output impact).

resources	architectures			
	A1 (Sec. 5.1)	A2 (Sec. 5.2)	A3 (Sec. 5.3)	A4 (Sec. 5.4)
`AddSub`	1	1	2	2
`Mult`	1	1	2	2
`CSWAP`	1 V1	1 V2	1 V2	1 V3
Data Memory	1	1	1	2
Communication System	1	1	1	2 with bridge
Program Memory	1	1	1	1
Control	1	1	1	1

Table 4. Main characteristics of the 4 implemented and evaluated architectures.

short name	model	techno. [nm]	slice content		f_{max}DSP [MHz]	BRAM capacity
			LUT	flip-flop		
V4	Virtex 4 VLX100	90	2 LUT4	2	500	18 Kb
V5	Virtex 5 LX110T	65	4 LUT6	4	550	36 Kb
S6	Spartan 6 SLX75	45	4 LUT6	4	390	18 Kb

Table 5. Target FPGAs with some characteristics (f_{max} is the maximum frequency).

(notice that a single `Mult` already handles 3 sets of operands in parallel using hyper-threading, see [9]). Architecture A4 is a cluster of parallel units for both arithmetic operations and data memory operations, and V3 of `CSWAP` unit.

The Xilinx FPGAs listed in Table 5 were our implementation targets. ISE 14.7 tools were used for synthesis and place&route, as well as SmartXplorer. V4/V5 FPGAs were used for comparison with state of the art. FPGA S6 was used for low-cost solutions and imminent SCA evaluation on SAKURA card [16]. In order to fairly compare area results, it should be remembered that slice and look-up table (LUT) definition strongly depends on the FPGA family, see examples in Table 5. Flip-flop (FF) means a 1-bit register. One LUT6 is equivalent to 4 LUT4. Then slices in V4 or in V5/S6 should not be compared directly.

For architectures A1–4, we report below implementation results for `w34`, `w68`, `w136` configurations on V4, V5, S6 FPGAs using 100 SmartXplorer runs.

5.1 Architecture A1: Base Solution

Architecture A1, depicted in Figure 2, corresponds to a basic Harvard processor dedicated to the scalar multiplication derived from [24] with our modifications detailed in Section 3.2. This is the smallest accelerator with only one instance of each type of unit (`AddSub`, `Mult`, and `CSWAP`-V1 described in Section 4.1).

Architecture A1 was fully implemented in VHDL for the 3 widths $w \in \{34, 68, 136\}$ on 3 FPGAs {V4, V5, S6}. The corresponding results are reported in Table 6. Intensive VHDL simulations were used for validation.

Table 6 reveals a few trends:

- Width w has a small impact (at most 5 % reduction) on the clock cycles count since most of the time is spent into the single `AddSub` and `Mult`.

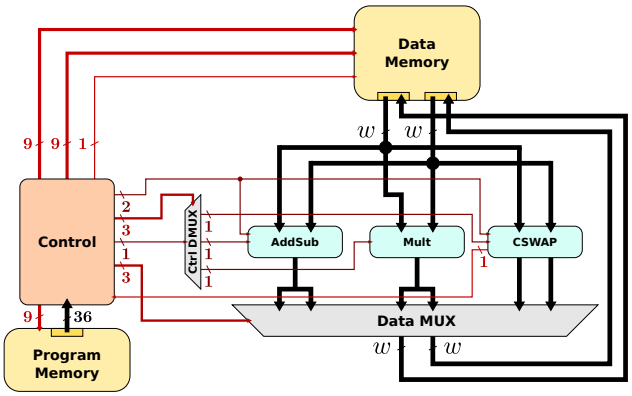

Fig. 2. Architecture A1 with its main units (arithmetic and memory), internal communication system, and control (warning: areas of boxes are not on a real scale).

- Width w strongly impacts the area in LUTs (+50–70 % for `w68` compared to `w34`, and +80–110 % for `w136` compared to `w34` depending on the FPGA). Clearly, enlarging the datapath requires more logic cells.
- The link between w and flip-flops numbers seems tricky. This is partly due to the cost of serial-parallel interfaces between arithmetic operators ($w_{\text{arith}} = 34$) and memory/communications (`w68` the most complex case, and `w136`).
- The increase in BRAMs comes from the wider memory for configurations `w68` and `w136` (see Table 2).
- Frequency decreases when w increases due to longer combinatorial delays and larger fanout. Depending on the FPGA, the reduction varies about 5–20 %. On V5, 360 MHz is the frequency for both the slowest unit and the complete accelerator (the control does not impact the overall frequency for a small accelerator which uses a small part of the complete FPGA).

FPGA	w [bit]	LUT	FF	logic slices	DSP blocks	RAM blocks	freq. [MHz]	clock cycles	time [ms]
V4	34	1010	1833	1361	11	4	322	194,614	0.60
	68	1750	3050	2251	11	5	305	186,911	0.61
	136	2281	3028	1985	11	7	266	184,337	0.69
V5	34	757	1816	603	11	4	360	194,614	0.54
	68	1264	3033	908	11	5	360	186,911	0.52
	136	1582	3008	940	11	7	360	184,337	0.51
S6	34	1064	1770	408	11	4	278	194,614	0.70
	68	1555	2970	705	11	5	252	186,911	0.74
	136	1910	2994	747	11	7	221	184,337	0.83

Table 6. FPGA implementation results for architecture A1 (all BRAMs are 18 Kb ones, only 17×17 multipliers were used in DSP slices for all FPGAs).

As a conclusion for our smallest architecture, using large w is not interesting. The reduction of the clock cycles count is canceled by the frequency drop for `w68` or `w136`. Hence, the best solution is always `w34` for A1 on all tested FPGAs.

5.2 Architecture A2: `CSWAP` Optimization

Architecture A2 is similar to A1 where we modified the `CSWAP` unit, version V2 (the architecture schematic is the same as Figure 2). The ML algorithm proposed in [24] uses one `CSWAP` operation at the end of each iteration and another `CSWAP` at the beginning of the next iteration with the same key bit operands. As there is no computation between these 2 consecutive `CSWAP` operations, we propose to merge them (this halves the calls to the `CSWAP` unit).

Our *modified* `CSWAP`-V2 uses 2 consecutive key bits: k_i and k_{i-1} (scalar k is used starting MSB first). There is no swapping when $k_i = k_{i-1}$, and swapping when $k_i \neq k_{i-1}$ (we just need one `xor` gate). The very first `CSWAP`-V2 call is computed using bits 0 (current bit) and k_{m-1} ("next" bit and MSB of k). The proposed modification does not change security aspects against SCAs. The accelerator is still constant-time and uniform. As for `CSWAP` in A1, we designed `CSWAP`-V2 with a uniform activity pipeline (see `CSWAP` description in Section 4.1).

Complete implementation results for A2 are reported in Table 7. A2 shows a similar behavior than A1 with respect to w variations. A few elements can be noticed for A2 as summarized below:

- Clock cycles count in A2 is slightly smaller than A1 due to reduced number of `CSWAP` operations.
- Frequency is slightly higher for large w compared to A1. Frequency variations are smaller in A2 than A1.
- Computation time in A2 is slightly smaller than A1: -5–10 % depending on the FPGA. The best solution is obtained for small w (the 0.8 % speed-up for `w68`/`w136` in V5 is not relevant due to the large area increase).
- Area (LUTs, FFs and slices) in A2 is slightly smaller than A1 due to a simplified management of `CSWAP` operations: -5–13 % depending on the FPGA.

FPGA	w [bit]	LUT	FF	logic slices	DSP blocks	RAM blocks	freq. [MHz]	clock cycles	time [ms]
V4	34	872	1624	1121	11	4	330	184,374	0.56
	68	1556	2637	1978	11	5	290	183,071	0.63
	136	2161	3027	2100	11	7	327	183,057	0.56
V5	34	722	1605	541	11	4	360	184,374	0.51
	68	1196	2620	840	11	5	360	183,071	0.51
	136	1419	3009	944	11	7	360	183,057	0.51
S6	34	940	1559	381	11	4	293	184,374	0.63
	68	1503	2565	553	11	5	262	183,071	0.70
	136	1890	2981	667	11	7	283	183,057	0.65

Table 7. FPGA implementation results for architecture A2 (all BRAMs are 18 Kb ones, only 17×17 multipliers were used in DSP slices for all FPGAs).

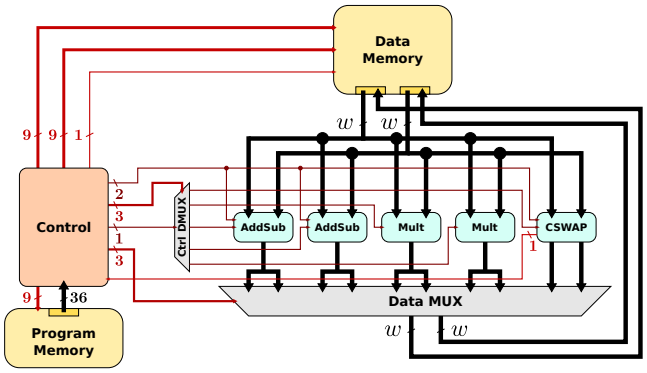

Fig. 3. Architecture A3 with its main units (arithmetic and memory), internal communication system, and control (warning: areas of boxes are not on a real scale).

– DSP slices and BRAMs are identical in A2 and A1 (not related to `CSWAP`-V2).

As a conclusion for architecture A2, the best configuration is always `w34` for all tested FPGAs. This optimization is interesting since A2 is slightly more efficient than A1 in terms of both speed and area (about 10 %).

5.3 Architecture A3: Large Architecture

Architecture A3, depicted in Figure 3, embeds more arithmetic units: 2 `AddSub` and 2 `Mult` units. It can perform up to 6 $\mathbb{F}_p$ multiplications in parallel (only 3 for A1/A2) using our hyper-threaded `Mult` unit (see Section 4.1 and [9]).

Complete implementation results for A3 are reported in Table 8. A3 behaves quite differently from A1/A2.

FPGA	w [bit]	LUT	FF	logic slices	DSP blocks	RAM blocks	freq. [MHz]	clock cycles	time [ms]
V4	34	1462	2611	1783	22	6	294	188,218	0.64
	68	2802	4367	3468	22	7	282	124,191	0.44
	136	3768	5017	3660	22	9	285	119,057	0.42
V5	34	1262	2607	921	22	6	358	188,218	0.53
	68	2290	4403	1409	22	7	345	124,191	0.36
	136	2737	4978	1594	22	9	348	119,057	0.34
S6	34	1527	2503	668	22	6	265	188,218	0.71
	68	2421	4267	1020	22	7	225	124,191	0.55
	136	3007	4877	1131	22	9	225	119,057	0.53

Table 8. FPGA implementation results for architecture A3 (all BRAMs are 18 Kb ones, only 17×17 multipliers were used in DSP slices for all FPGAs).

– Adding 1 `AddSub` and 1 `Mult` increases LUTs by 60–90 % depending on the FPGA and w. The second `Mult` adds 11 DSP slices and 2 BRAMs. This confirms that $\mathbb{F}_p$ units constitute the largest resources in the accelerator.
– Frequency is slightly smaller in A3 than A2 due to larger fanout and more complex control. The frequency drop for increasing w values is very small for V4/V5 (less than 4 %), and about 15 % for S6.
– Unlike A1/A2, w has a large impact on the clock cycles count in A3: 34 % reduction for `w68` compared to `w34` and 36 % for `w136`. More arithmetic operations in parallel put pressure on the memory and communication system. A larger w allows to actually exploit more parallelism.
– Computation time benefits from the reduction of clock cycles count for large values of w: 25 to 35 % reduction for `w136` depending on the FPGA.
– A3 is faster than A2: from 16 to 35 % depending on the FPGA. But this speed-up comes at the expense of a larger area.

As a conclusion for architecture A3, there is no best solution but various compromises in terms of area and speed. When area is limited, `w34` is interesting but computation time is 25–33 % larger. When speed is the main objective, `w68` and `w136` lead to the fastest solutions but the area overhead is important.

5.4 Architecture A4: Clustered Architecture

A closer look at Figure 1 shows that `xDBLADD` can be decomposed into 2 clusters of $\mathbb{F}_p$ operations with few dependencies using the red dashed horizontal line (*i.e.* one cluster above, one cluster below). Only 4 values have to be transferred from bottom cluster to top one at each ML iteration.

Architecture A4, depicted in Figure 4, was designed to exploit this decomposition using a clustered accelerator. It also embeds a new optimization of the `CSWAP` unit described below. To lighten Figure 4, control signals are not completely drawn but represented by small circles. Constants values are duplicated in each cluster memory when necessary (at no cost in BRAMs).

We added a new modification of the `CSWAP` behavior (V3). `CSWAP` operation is replaced by 2 new swapping operations $\texttt{CS}_0$ and $\texttt{CS}_1$ (with 4 $\mathbb{F}_p$ operands):

– $\texttt{CS}_0(A, B, C, D)$ returns (A, B, C, B) if $k_i = 0$, else it returns (C, D, A, D)
– $\texttt{CS}_1(A, B, C, D)$ returns (A, B, C, D) if $k_i = 0$, else it returns (C, D, A, B).

The modification of the `CSWAP` behavior (V3) was associated to a new schedule for `xDBLADD` presented in Figure 5. This figure is simplified, each black line now represents the communication of 4 operands from Figure 1. `H` box represents a set of 8 $\mathbb{F}_p$ additions/subtractions, and `M` box a set of 4 $\mathbb{F}_p$ multiplications (only 3 in the top right upper box of Figure 5). $\texttt{CS}_0$ and $\texttt{CS}_1$ are respectively in charge of the first and last `CSWAP`-V3 of the original ML iteration. Square operations of the original `xDBLADD` have been replaced by multiplications in the new solution (*i.e.* $A^2 \times B$ is now $A \times B \times A$) since we do not implement dedicated square units. This does not change the mathematical behavior nor the operations count compared to A2/A3, but it allows to use A4 more efficiently. $\texttt{CS}_0$ and $\texttt{CS}_1$ operations also

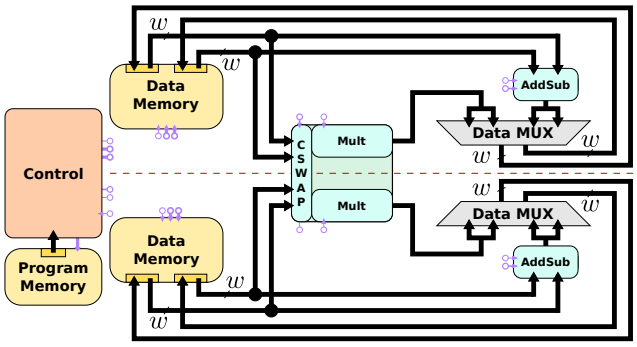

Fig. 4. Architecture A4 with its 2 clusters (units, data memory and local communication system), common control and new modification of CSWAP (warning: areas of boxes are not on a real scale).

act as "bridge" to exchange data between the 2 clusters when shared into a single `CSWAP`-V3 unit as illustrated in Figure 4.

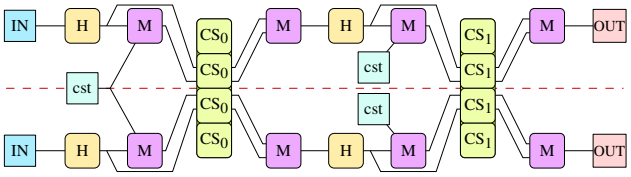

Fig. 5. Modified `xDBLADD` formula for architecture A4 with the new `CSWAP` behavior.

Architecture A4 (Figure 4) is based on two identical clusters. Each cluster contains: 1 `AddSub`, 1 `Mult`, 1 data memory and 1 local communication system. The shared `CSWAP`-V3 unit uses a 2-bit control mode to select the relevant swapping pattern according to the dependency graph in Figure 5. The control is shared for the 2 clusters, management of ML iterations and external inputs/outputs (at beginning/end of scalar multiplication).

Complete implementation results for A4 are reported in Table 9. A few elements are summarized below:

- Clock cycles count is reduced for `w34` but slightly increased for `w68` and `w136` compared to A3. This is due to additional constraints on the scheduler for A4 since all units do not share a common memory and clusters can only exchange data during the new modified `CSWAP`-V3.
- Frequency is higher in A4 compared to A3 in most of case (and very similar in the other cases). This is due to a more local control and a smaller fanout (most of signals are local to each cluster).

FPGA	w [bit]	LUT	FF	logic slices	DSP blocks	RAM blocks	freq. [MHz]	clock cycles	time [ms]
V4	34	1695	2950	2158	22	7	324	142,119	0.44
	68	2804	4282	3184	22	9	290	128,021	0.44
	136	3171	4994	3337	22	13	299	125,456	0.42
V5	34	1370	2953	1013	22	7	358	142,119	0.40
	68	2095	4259	1358	22	9	337	128,021	0.38
	136	2514	4952	1589	22	13	313	125,456	0.40
S6	34	1564	2089	758	22	7	262	142,119	0.54
	68	2387	4030	1060	22	9	239	128,021	0.54
	136	3181	4786	1136	22	13	251	125,456	0.50

Table 9. FPGA implementation results for architecture A4 (all BRAMs are 18 Kb ones, only 17×17 multipliers were used in DSP slices for all FPGAs).

- Computation time is significantly reduced for small w values. For instance, `w34` leads to a similar speed than A3 but with much smaller architecture.
- On V5, the fastest solution is the intermediate configuration `w68`.
- DSP slices amount is exactly the same in A4 and A3 (both use 2 `Mult` units).
- BRAMs amount is larger in A4 than A3 due to the 2 local memories (one in each cluster). The number of BRAMs increases with w accordingly to Table 2 configurations.
- Area results in terms of LUTs are quite different from previous architectures. It increases for `w34` (up to +15 %) but decreases for `w136` (up to -16 %) compared to A3.
- Adding a second memory unit allows parallel read/write accesses and helps to quickly extract more parallel operations.

As a conclusion for architecture A4, selecting the right set of parameters depends a lot on the objective (high speed or low cost) and the FPGA family. When the main objective is selecting the absolute smallest accelerator, A4 is less interesting then A3. When the main objective is selecting the absolute fastest accelerator, A4 is interesting for low-cost S6 FPGA (but A3 is better for V4/V5). But in practice, A4 is interesting for accelerators almost as fast as the fastest A3 but for much smaller area. For instance on V4, `w34` configuration is only 5 % slower than the absolute fastest A3 solution with an area reduced by 55 % in LUTs and 22 % in BRAMs.

6 Comparisons

Figure 6 reports all the implementation results for the 4 proposed architectures and w configurations on 3 different FPGAs. This figure only reports LUTs for area since there are only a very few different numbers of BRAMs and DSP slices given in Table 10. The best configuration (architecture type, w) depends on the objective (high speed or low area) and target FPGA. Then exploration tools at architecture level are helpful for designers and users. For low-area solution,

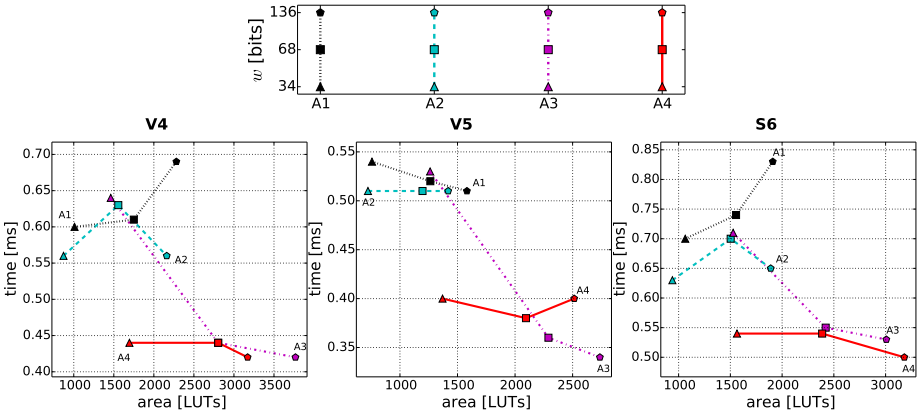

Fig. 6. Trade-offs for our architectures A1–4 in terms of area (LUTs) and computation time for all w configurations and FPGAs (legend is top figure).

archi.	w [bit]	target	logic slices	DSP blocks	RAM blocks	freq. [MHz]	time [ms]
A2	34	V4	1121	11	4	330	0.56
A3	136		3660	22	9	285	0.42
A4	34		2158	22	7	324	0.44
A2	34	V5	541	11	4	360	0.51
A3	136		1594	22	9	348	0.34
A4	34		1013	22	7	358	0.40
A2	34	S6	381	11	4	293	0.63
A3	136		1131	22	9	225	0.53
A4	34		758	22	7	262	0.54

Table 10. Summary of our most interesting FPGA implementation results (those on Pareto front from Figure 6).

A2 is always the best one. For high speed, A4 is a very good cost-performance trade-off on V4 and S6 (A3 is the fastest on V5 but with a large area).

Most of hardware HECC implementations use curves over $\mathbb{F}_{2^n}$ with low security levels (typically 81–89 bits fields). Table 11 reports some of them. None of those HECC implementations embed hardware level protections against SCAs. Some of them use algorithmic protections for scalar multiplication such as the Montgomery ladder. Our $\mathbb{F}_p$ accelerators show similar computation times but for a much higher security level (128 bits) on more recent FPGAs. To the best of our knowledge, we found only one hardware implementation of HECC over $\mathbb{F}_p$ in [1]. It is an 0.13 μm ASIC implementation for 81-bit generic prime p with 502.8 ms computation time for scalar multiplication at 1 MHz. The very low reported frequency makes comparisons quite difficult.

ref.	year	target	n	LUT	FF	logic slices	RAM blocks	freq. [MHz]	time [ms]
[3]	2006	Virtex 2 Pro	83	20999	n.a.	11296	n.a.	166	0.5
[8]	2008	XC2V4000	83	n.a.	n.a.	2316	6	125	0.31
[13]	2004	XC2V4000	89	8451	2178	4995	1	54	1.02
		XC2V4000	89	16459	4437	9950	0	57	0.44
[25]	2006	Virtex 2 Pro	83	n.a.	n.a.	2446	1*	100	0.99
		Virtex 2 Pro	83	n.a.	n.a.	6586	3*	100	0.42
[26]	2016	Virtex 2	83	n.a.	n.a.	5734	n.a.	145	0.3
		XC5V240	83	n.a.	n.a.	5086	n.a.	175	0.29
[27]	2004	Virtex 2 Pro	81	n.a.	n.a.	4039	1	57	0.79
		Virtex 2 Pro	81	n.a.	n.a.	7737	0	61	0.39
		XC2V4000	81	n.a.	n.a.	3955	1	54	0.83
		XC2V4000	81	n.a.	n.a.	7785	n.a.	57	0.42
[7]	2007	XC2V8000	113	n.a.	n.a.	25271	n.a.	45	2.03

Table 11. FPGA implementation results for various HECC solutions over $\mathbb{F}_{2^n}$ from state of the art (warning: security levels are much lower than our solutions). For $\mathbb{F}_{2^n}$ DSP slices cannot be used. Values with a "*" are estimated number of RAM blocks based on paper explanations.

ref.	year	target	p	LUT	FF	logic slices	DSP blocks	RAM blocks	freq. [MHz]	time [ms]
[2]	2014	XCV6FX760	NIST-256	32900	n.a.	11200	289	128	100	0.4
[11]	2008	XC4VFX12	NIST-256	2589	2028	1715	32	11	490	0.5
		XC4VFX12	NIST-256	34896	32430	24574	512	176	375	0.04
[17]	2012	XC4VFX12	GEN-256	n.a.	n.a.	2901	14	n.a.	227	1.09
		XC5VLX110	GEN-256	n.a.	n.a.	3657	10	n.a.	263	0.86
[19]	2013	XC4VLX100	GEN-256	5740	4876	4655	37	11	250	0.44
		XC5LX110T	GEN-256	4177	4792	1725	37	10	291	0.38

Table 12. FPGA implementation results for various ECC solutions over $\mathbb{F}_p$ and 128-bit security level from state of the art.

Directly comparing our accelerators with implementations of HECC over $\mathbb{F}_{2^n}$ for much lower security level is not possible. Then, in Table 12, we report some of the best FPGA implementations results we found in the state of the art for ECC solutions over $\mathbb{F}_p$ and 128 bits security level. In practice using hundred of DSP blocks and BRAMs may not be a realistic solution for embedded systems. In [2] several SCAs protections have been presented: DBL&ADD-always, ML, scalar randomization, units with uniform behavior, randomization of memory addresses and noise addition. Those protections impact the number of logic slices but not those of DSPs and BRAMs (very huge is this work).

Compared to [19], a very optimized fast and compact solution from state of the art using randomized Jacobian coordinates, our accelerator have a very similar computation time (0.44 ms) but with 40 % reduction in DSP and RAM blocks and 53 % reduction of logic slices on V4 FPGA (similar for V5).

7 Conclusion and Future Prospects

We proposed the first hardware implementation of Kummer based HECC solution for 128-bit security level. Various architectures and parameters have been explored using in-house tools. Several architectures with different amount of internal parallelism have been optimized and fully implemented on 3 different FPGAs. The obtained results lead to similar speed than the best curve based solutions for embedded systems but with an area almost divided by 2 (-40 % for DSP and RAM blocks and -60 % for logic slices). Those results were obtained with generic prime fields and fully programmable architectures (which is not the case in most of state of the art implementations).

In the future, we plan to optimize our tools and architectures, evaluate the security against SCAs using real measurement setup, automate the control generation of the accelerator, and publish our architectures as open source hardware.

Acknowledgment

This work was done in the HAH project `http://h-a-h.inria.fr/` partially funded by Labex CominLab, Labex Lebesgue and Brittany Region.

References

1. H.-R. Ahmadi, A. Afzali-Kusha, M. Pedram, and M. Mosaffa. Flexible prime-field genus 2 hyperelliptic curve cryptography processor with low power consumption and uniform power draw. *ETRI journal*, 37(1):107–117, February 2015.
2. H. Alrimeih and D. Rakhmatov. Fast and flexible hardware support for ECC over multiple standard prime fields. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12):2661–2674, January 2014.
3. L. Batina, N. Mentens, B. Preneel, and I. Verbauwhede. Flexible hardware architectures for curve-based cryptography. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 4839–4842. IEEE, May 2006.
4. D. J. Bernstein and T. Lange. Explicit-formulas database. `http://hyperelliptic.org/EFD/`.
5. J. W. Bos, C. Costello, H. Hisil, and K. Lauter. Fast cryptography in genus 2. *Journal of Cryptology*, 29(1):28–60, January 2016.
6. H. Cohen and G. Frey, editors. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Discrete Maths and Applications. Chapman & Hall/CRC, July 2005.
7. G. Elias, A. Miri, and T.-H. Yeap. On efficient implementation of FPGA-based hyperelliptic curve cryptosystems. *Computers & Electrical Engineering*, 33(5):349–366, July 2007.
8. J. Fan, L. Batina, and I. Verbauwhede. HECC goes embedded: an area-efficient implementation of HECC. In *Proc. 15th Workshop on Selected Areas in Cryptography (SAC)*, volume 5381 of *LNCS*, pages 387–400. Springer, August 2008.
9. G. Gallin and A. Tisserand. Hyper-threaded multiplier for HECC. In *Proc. 51st Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, USA, October 2017. IEEE.

10. P. Gaudry. Fast genus 2 arithmetic based on theta functions. *Journal of Mathematical Cryptology*, 1(3):243–265, August 2007.
11. T. Güneysu and C. Paar. Ultra high performance ECC over NIST primes on commercial FPGAs. In *Proc. 10th Conf. Cryptographic Hardware and Embedded Systems (CHES)*, volume 5154 of *LNCS*, pages 62–78. Springer, August 2008.
12. D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.
13. H. Kim, T. J. Wollinger, Y. Choi, K. Chung, and C. Paar. Hyperelliptic curve coprocessors on a FPGA. In *Proc. 5th International Workshop on Information Security Applications (WISA)*, volume 3325 of *LNCS*, pages 360–374. Springer, August 2004.
14. N. Koblitz. Hyperelliptic cryptosystems. *Journal of Cryptology*, 1(3):139–150, October 1989.
15. C. K. Koc, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
16. Satoh Lab. and Morita Tech. Side-channel attack user reference architecture (SAKURA), 2013.
17. J.-Y. Lai, Y.-S. Wang, and C.-T. Huang. High-performance architecture for elliptic curve cryptography over prime fields on FPGAs. *Interdisciplinary Information Sciences*, 18(2):167–173, 2012.
18. T. Lange. Formulae for arithmetic on genus 2 hyperelliptic curves. *Applicable Algebra in Eng., Communication and Computing*, 15(5):295–328, February 2005.
19. Y. Ma, Z. Liu, W. Pan, and J. Jing. A high-speed elliptic curve cryptographic processor for generic curves over GF(p). In *Proc. 20th International Workshop on Selected Areas in Cryptography (SAC)*, volume 8282 of *LNCS*, pages 421–437, Burnaby, BC, Canada, August 2013. Springer.
20. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.
21. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
22. P. L. Montgomery. Speeding the pollar and elliptic curves methods of factorisation. *Mathematics of Computation*, 48(177):243–264, January 1987.
23. H. Orup. Simplifying quotient determination in high-radix modular multiplication. In *Proc. 12th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 193–199, Bath, UK, July 1995. IEEE.
24. J. Renes, P. Schwabe, B. Smith, and L. Batina. μKummer: Efficient hyperelliptic signatures and key exchange on microcontrollers. In B. Gierlichs and A. Y. Poschmann, editors, *Proc. 18th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, volume 9813 of *LNCS*, pages 301–320, Santa Barbara, CA, USA, August 2016. Springer.
25. K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. Superscalar coprocessor for high-speed curve-based cryptography. In *Proc. 8th Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 4249 of *LNCS*, pages 415–429. Springer, October 2006.
26. A. Sghaier, C. Massoud, M. Zeghid, and M. Machhout. Flexible hardware implementation of hyperelliptic curves cryptosystem. *International Journal of Computer Science and Information Security (IJCSIS)*, 14(4), April 2016.
27. T. Wollinger. *Software and hardware implementation of hyperelliptic curve cryptosystems*. Ruhr University Bochum, 2004.