An 8-bit AVR-Based Elliptic Curve Cryptographic RISC Processor for the Internet of Things

Erich Wenger
Graz University of Technology
IAIK, Inffeldgasse 16a, A–8010 Graz, Austria
erich.wenger@iaik.tugraz.at

Johann Großschädl
University of Luxembourg
6, r. Coudenhove-Kalergi, L–1359 Luxembourg
johann.groszschaedl@uni.lu

***Abstract*—In recent years, a large body of research has been dedicated to the "lightweight" implementation of Elliptic Curve Cryptography (ECC) for RFID tags, wireless sensor nodes, and other "smart" devices that are supposed to become components of the Internet of Things (IoT). However, making ECC suitable for the IoT is far from trivial since many applications demand fast response times (i.e. high performance), but nonetheless call for small silicon area and low power consumption. We tackle this challenge through hardware/software co-design and introduce an 8-bit Application-Specific Instruction Set Processor (ASIP) that combines the efficiency of a dedicated hardware implementation with the flexibility and scalability of ECC software. Our ASIP is based on JAAVR ("Just Another AVR"), an ATmega128 clone into which we integrated a relatively small (32×4)-bit Multiply-Accumulate (MAC) unit optimized to speed up multi-precision arithmetic. To demonstrate the flexibility of our co-design, we implemented scalar multiplication over four families of elliptic curve, namely a Weierstraß curve, a twisted Edwards curve, a Montgomery curve, and a Gallant-Lambert-Vanstone curve. All curves use a 160-bit Optimal Prime Field (OPF) as underlying algebraic structure, which allows for particularly fast execution of the modular reduction on JAAVR. When using "native" AVR instructions only, our fastest implementation of scalar multiplication reaches an execution time of less than 4M clock cycles on a conventional ATmega128 processor. Taking advantage of the MAC unit, the time for a full 160-bit scalar multiplication falls below 1M cycles, whereas a "leakage-reducing" implementation that does not execute any security-critical conditional statements needs some 1.3M cycles. A low-memory variant of the extended JAAVR occupies an area of merely 21k gates, making it suitable for resource-constrained IoT devices like sensor nodes.**

***Index Terms*—Lightweight cryptography, IoT security, Wireless sensor network, AVR architecture, Instruction set extension**

I. Introduction

The so-called "Internet of Things" (IoT) is envisioned as a global network connecting billions of smart devices capable of creating, processing, and/or exchanging data without the intervention of human beings. Today, virtually all products that use electricity, ranging from home appliances (e.g. coffee machines, fridges, micro-waves) over entertainment systems (e.g. TV sets, DVD players, game consoles) to cars (or, more generally, smart vehicles) are equipped with microprocessors and, consequently, possess data processing capabilities. An ever-increasing number of these devices contain transceivers for WiFi, GSM/GPRS, Bluetooth, Zigbee, or other wireless networking technologies, which allows them to communicate with each other or establish a connection to the Internet. In addition, also "non-smart" things (e.g. consumer products in an ordinary retail store) can temporarily become a part of the IoT by attaching RFID tags to them so that they evolve into identifiable objects.

Security and privacy issues pose a significant challenge to the further expansion of the IoT and the end-user acceptance of many IoT-based applications and services [18]. Similar to the "ordinary" Internet, Public-Key Cryptography (PKC) can play a valuable role in the IoT to overcome these challenges by providing such services as encryption, authentication, and key establishment. *Elliptic Curve Cryptography (ECC)* [10] is a form of PKC that offers equivalent security to RSA (and other "classical" PKC schemes), but does so with significantly shorter keys. A well-designed 160-bit ECC cryptosystem is presumed to be at least as secure as 1024-bit RSA [10]. The high level of security per bit of ECC implicates a multitude of further benefits, most notably fast computation time, small RAM footprint, and low bandwidth requirements, which has made ECC a serious competitor to RSA. All the advantages of ECC over RSA are even more pronounced in the IoT since sensor nodes, RFID tags, and the like are extremely limited in computational power and memory resources.

The efficient implementation of ECC for IoT devices is an intricate task due to diverse (and even conflicting) constraints and requirements that need to be taken into account. On the one hand, many IoT applications require fast response times (i.e. "high" performance), while, at the same time, they also demand small silicon area and low power dissipation. These requirements make a good case for hardware implementation of the expensive operations of ECC. On the other hand, the main building block of ECC (namely scalar multiplication in an elliptic curve group [10]) is very complicated, which calls for a software solution. Furthermore, ECC is a highly active area of research that yields better and better algorithms and parameterizations every year, and this progress also benefits implementation results. Adapting an ECC implementation in response to mathematical progress is only possible with software since an algorithm cast in silicon can not be modified or updated. A promising approach to cope with such diverse constraints is hardware/software co-design in the form of an Application-Specific Instruction Set Processor (ASIP).

In this paper, we describe the design of an area-optimized 8-bit ASIP for ECC and assess the execution time of several scalar multiplication algorithms executed on it. Our ASIP is

Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture Workshops (MICROW 2012), pp. 39–46.

based on JAAVR ("Just Another AVR"), an AVR-compatible softcore written in synthesizeable VHDL [24], into which we integrated a special Multiply-Accumulate (MAC) unit so as to speed up long-integer arithmetic [15]. JAAVR is basically an ATmega128 clone consisting of only 6,166 GE that supports the "full" AVR instruction set. The MAC unit of our ASIP is composed of a (32×4)-bit multiplier and a 72-bit accumulator. It is tightly integrated into JAAVR's datapath and operates on General-Purpose Register (GPR), i.e. the 32-bit operands to be multiplied are read from two quadruples of 8-bit GPRs (32 bits per operand). A full (32×32)-bit multiplication takes eight clock cycles and does not stall the pipeline of JAAVR's integer unit. The 64-bit product is then accumulated into the content of the nine GPRs `R0` to `R8`. The idea of integrating a functional unit able to process operands which are a multiple of the processor's native word-size is somewhat related to the rationale of many multimedia extensions (e.g. the 128-bit SSE extensions to the 32-bit Intel Architecture), but we apply this idea to accelerate ECC on an 8-bit processor.

Hardware acceleration of ECC at the granularity of custom instructions has the advantage of high flexibility and scalability. To demonstrate this, we implemented scalar multiplication using four different families of elliptic curve: a conventional Weierstraß curve [10], a Montgomery curve [20], a so-called twisted Edwards curve [12] and a Gallant-Lambert-Vanstone (GLV) curve [6]. All the curves use a 160-bit Optimal Prime Field (OPF) [26] as underlying algebraic structure. OPFs are a special class of prime fields that allow for particularly efficient arithmetic on our ASIP since they perform the reduction operation through multiplications (and not via additions like generalized-Mersenne primes), which are very fast thanks to the (32×4)-bit MAC unit. We also evaluated the execution time of the standardized Weierstraß curve `secp160r1` to have a reference for comparison. For each curve, we implemented the field arithmetic in two alternative ways, namely with and without MAC operation. Our fastest implementation is based on the GLV curve and performs a 160-bit scalar multiplication in slightly less than 4M clock cycles when using only native AVR instructions. By taking advantage of the MAC unit, the execution time can be reduced to 1M cycles. However, this implementation is vulnerable to side-channel attacks due to the "irregular" execution pattern of the GLV method. On the other hand, scalar multiplication on the Montgomery curve features regular execution, but is a little slower since it needs 5.5M cycles on an ordinary ATmega128 and 1.3M cycles on our ASIP. The former result improves the state-of-the-art in constant-time scalar multiplication on an 8-bit processor.

II. Preliminaries

In this section we summarize basic concepts and properties of the algebraic structures used in our work, namely Optimal Prime Fields (OPFs) [26] and three special classes of curves (Montgomery curves [20], twisted Edwards curves [12], and GLV curves [6]). We will not discuss Weierstraß curves since they are described in any textbook on ECC, e.g. in Hankerson et al's Guide to Elliptic Curve Cryptography [10].

A. Optimal Prime Fields

The finite field we use in our implementation belongs to the class of so-called *Optimal Prime Fields (OPFs)*, which were first mentioned in the literature in an extended abstract from 2006 [7]. These fields are specified by a "low-weight" prime that can be written as $p = u \cdot 2^k + v$, where u and v are small compared to 2^k; in our case, u has a length of up to 16 bits so that it fits into two registers of an ATmega processor, while v is equal to 1. Primes of such a form are characterized by a low Hamming weight as only the two most significant bytes and the least significant byte are non-zero; all other bytes are zero. An example is $p = 65356 \cdot 2^{144} + 1$, which happens to be a 160-bit prime that looks as follows in hex notation.

0xFF4C0000000000000000000000000000000000000001

The low Hamming weight of p allows for optimization of the modular arithmetic since only the non-zero digits of p need to be processed in the reduction operation. For example, the Montgomery algorithm [19] can be easily optimized for these primes such that the modular reduction has only linear complexity, similar to pseudo-Mersenne or generalized-Mersenne primes [10]. However, the latter family of primes allows one to do modular reduction via additions, whereas the reduction modulo our low-weight prime is performed through execution of `mul` instructions (resp. MAC instructions on our ASIP). As a consequence, OPFs are especially efficient on our ASIP and other platforms featuring a fast integer multiplier. A detailed description of how OPF arithmetic can be implemented on an 8-bit AVR processor is given in Section III.

B. Montgomery Curves

25 years ago, Peter Montgomery [20] introduced a special family of curves to speed up algorithms for the factorization of large integers. These curves are nowadays often referred to as Montgomery curves and have the unique property that a scalar multiplication can be carried out using the x coordinate only, which is significantly faster than when both the x and y coordinate are calculated. Formally, a Montgomery curve over $\mathbb{F}_p$ is defined by the equation

$$By^2 = x^3 + Ax^2 + x \tag{1}$$

with $(A^2 - 4)B \neq 0$ and allows for an efficient computation of the x-coordinate of a sum $P+Q$ of two points P, Q whose difference $P-Q$ is known. More precisely, a point addition performed according to the formula in [20, p. 261] requires four multiplications (4M) and two squarings (2S), whereas a point doubling costs 3M and 2S. However, one of the three multiplications in the doubling uses the constant $(A+2)/4$ as operand, which is small if the curve parameter A is chosen accordingly. Our experiments show that multiplying a field element by a small (up to 16 bits long) constant costs some 0.25–0.3 M. Furthermore, the point addition formula specified in [20, page 261] can be optimized when using the so-called Montgomery ladder [3, Alg. 13.35] for scalar multiplication and representing the base point P via projective coordinates

with $Z = 1$ (see also Remark 13.36 (ii) in [3]). Although the number of field multiplications and squarings is low, it has to be taken into account that the Montgomery ladder always executes both a point addition and a point doubling for each bit of the scalar k. For this reason, the overall computational cost of scalar multiplication amounts to $5.3n$ multiplications and $4n$ squarings in $\mathbb{F}_p$, i.e. 5.3 M + 4 S per bit.

C. Twisted Edwards Curves

In 2007, Harold Edwards introduced a new form of elliptic curves that are now known as Edwards curves [4]. Bernstein et al [2] discussed these curves in more detail and presented an especially fast implementation of point addition and point doubling. Moreover, they showed that these curves allow one to define an addition law with very attractive implementation properties such as uniformity (i.e. it can be used for addition as well as doubling of points) and completeness (i.e. it works for any input, including the point at infinity). Twisted Edwards curves, also proposed by Bernstein et al, are a generalization of Edwards curves and can be defined by an equation of the following form:

$$ax^2 + y^2 = 1 + dx^2y^2 \tag{2}$$

where a and d are distinct elements of $\mathbb{F}_p \setminus \{0\}$. We use the extended twisted Edwards coordinates from [12] to perform addition and doubling of points; the former operation requires 7 multiplications (7M) in the underlying field $\mathbb{F}_p$, whereas the latter has a computational cost of 3 multiplications (3M) and 4 squarings (4S). When using the double-and-add method, a scalar multiplication $k \cdot P$ on a twisted Edwards curve takes $6.5n$ multiplications and $4n$ squarings in $\mathbb{F}_p$, i.e. 6.5 M + 4 S per bit, assuming that k has a Hamming density of 0.5.

D. GLV Curves

The so-called Gallant-Lambert-Vanstone (GLV) curves are elliptic curves over $\mathbb{F}_p$ that possess an efficiently computable endomorphism ϕ whose characteristic polynomial has small coefficients [6]. The specific curve used in this paper belongs to the family of GLV curves that can be described through a Weierstrass equation of the form

$$y^2 = x^3 + b \quad \text{(i.e. } a = 0\text{)} \tag{3}$$

over a prime field $\mathbb{F}_p$ with $p \equiv 1 \bmod 3$ (cf. Example 4 from [6]). When using mixed Jacobian-affine coordinates, a point addition on this curve requires 8 M + 3 S, i.e. adding points is exactly as costly as on a conventional Weierstrass curve. On the other hand, the double $2P$ of a point P given in Jacobian coordinates can be computed using only 3 M and 4 S since the curve parameter a is 0. However, what makes GLV curves very attractive is that the cost of scalar multiplication can be significantly reduced by exploiting the efficiently-computable endomorphism described in [6]. This endomorphism allows one to accomplish an n-bit scalar multiplication $k \cdot P$ through a computation of the form $k_1 \cdot P + k_2 \cdot \phi(P)$ where k_1, k_2 have only half the bit-length of k. The two half-length scalar multiplications can be carried out simultaneously (via "Shamir's trick"), which takes $n/2$ doublings and roughly $n/4$ additions when the scalars k_1, k_2 are represented in Joint Sparse Form (JSF) [10]. Thus, the overall cost of computing $k \cdot P$ amounts to $3.5n$ multiplications and $2.75n$ squarings in $\mathbb{F}_p$, i.e. 3.5 M + 2.75 S per bit. Note that the GLV method does not require the point P to be fixed or known a-priori, which means it can be used in ECDH key exchange.

III. Baseline Implementation

Our previous work [26] introduces an efficient OPF arithmetic library for 8-bit AVR processors and explains how to speed up modular reduction for low-weight primes. All ECC implementations we describe in this paper use an optimized version of the OPF library from [26] as a component for the "low-level" field arithmetic. However, for the sake of completeness, we provide in this section an overview of how the library performs addition, subtraction, multiplication, as well as squaring in OPFs. Throughout this paper, we will use the following notation. Uppercase letters denote arrays of w-bit words representing elements of $\mathbb{F}_p$, while indexed uppercase letters refer to individual words within an array, e.g. A_i is the i-th word of array A that represents $a \in \mathbb{F}_p$. Although AVR is an 8-bit platform, we use a word-size of $w = 32$ bits so as to increase performance, which means the arithmetic operations of our library generally process four bytes at once [9]. Given an operand length of n bits, the total number of w-bit words is $s = \lceil n/w \rceil$, i.e. we have $s = 5$ for a 160-bit OPF.

A. Addition and Subtraction

Addition and subtraction are the most basic operations in multiple-precision arithmetic. To calculate the modular sum $a + b \bmod p$, we firstly do the addition and then perform the reduction. Let A_i, B_i be the i-th word of the two arrays A and B, which represent $a, b \in \mathbb{F}_p$. The addition simply starts with $A_0 + B_0$, and then repeatedly calculates $A_i + B_i + c$ for i from 1 to $s-1$, whereby c denotes the carry bit generated in the previous addition of 32-bit words. After addition of the two most significant words, we have a sum that is up to $n+1$ bits long. We use the carry bit c from the last addition of words to decide whether or not to subtract p, which is faster than an exact comparison, but may lead to an incompletely reduced result in the range $[0, 2^n - 1]$ instead of the least non-negative residue. Fortunately, this is not a problem in practice because all arithmetic functions of our OPF library are able to cope with incompletely reduced operands.

A drawback of this approach is that, in the worst case, two subtractions of p are required to obtain an n-bit result since both operands may be incompletely reduced [26]. In order to get "branch-less" code, we always perform two subtractions of $c \cdot p$, but update the carry bit c after the first one. More in detail, the first subtraction produces a "borrow bit," which is either 0 or 1 and has to be subtracted from the carry bit to obtain a correct carry bit for the second subtraction. There exist a number of low-level optimization techniques to speed up these two final subtractions by exploiting the special form of p [26]. Note that the prime we use in this paper contains

$s-2$ zero-words; only the Most Significant Word (MSW) as well as the Least Significant Word (LSW) are not zero. All these zero-words do not need to be loaded from memory and normally do also not need to be subtracted from the sum. In other words, it normally suffices to perform the subtraction on the LSW and MSW of the sum. However, there is a case that requires special attention, namely when the LSW of the sum is 0 and the carry bit c is 1. In this situation, $c = 1$ gets subtracted from 0, which produces a "borrow" bit into the next-higher word. Our implementation tackles this issue by checking whether a borrow occurred, and if this is the case we propagate the borrow bit up to the MSW. Fortunately, the probability of a borrow being generated (which can only happen if the LSW of the sum $A+B$, is 0) is extremely low; in our case ($w = 32$), this probability is 2^{-32}. The information leakage from this (potential) irregularity is too small for a real SPA attack, in particular when the base point is blinded.

A modular substraction $a - b \bmod p$ is very similar to the modular addition, except that the prime p needs to be added if the difference $a - b$ is negative.

B. Multiplication and Squaring

Modular multiplication and squaring have a big impact on the overall performance of public-key cryptosystems, especially ECC [10]. Our OPF library uses Montgomery's algorithm [19] for modular multiplication and is optimized to achieve minimal execution time for "low-weight" primes of the form $p = u \cdot 2^{16} + 1$, whereby u has a length of at most 16 bits. An s-word array P representing p contains merely two non-zero words, namely P_{s-1} and P_0. There exist different techniques for fast computation of the Montgomery product; one is the so-called *Finely Integrated Product Scanning (FIPS)* method [14], which performs multiplication and modular reduction in an interleaved fashion. The FIPS method normally executes $2s^2 + s$ word-level (i.e. $(w \times w)$-bit) multiplications, but this number drops by roughly one half to $s^2 + s$ when the modulus is a low-weight prime as defined above [26]. Computing the product of two s-word operands requires s^2 word-level multiplications, which means the overhead of modular reduction is linear, costing only s word-level multiplications.

The FIPS method consists of two nested loops, both executing simple Multiply-Accumulate (MAC) operations in the inner loop. In our case (i.e. $w = 32$), the inner-loop operation requires multiplying two 32-bit words (which involves a total of 16 `MUL` instructions on an AVR processor) and adding the obtained 64-bit product to a 72-bit cumulative sum held in 9 registers. The concrete implementation of the FIPS method included the OPF library follows the general idea of hybrid multiplication [9], but executes the inner-loop operation in a more efficient way as described in [26]. An iteration of the inner loop needs only 101 clock cycles, which allows a full (160×160)-bit multiplication (without reduction) to be performed in 2,840 cycles. The execution of a 160-bit OPF-FIPS multiplication (including Montgomery reduction) takes some 3,314 clock cycles, i.e. our optimized OPF library is a little faster than Zhang's original version from [26]. Note that we exploit the same "shortcut" for the final subtraction as in the modular addition, which means we perform the subtraction only at the MSW and LSW. Similar to the modular addition described before, there is a small probability for information leakage from a (potential) irregularity in the subtraction.

IV. ECC Extensions for AVR

The ATmega128 is one of the most widely-used embedded processors of all time [1]. It is a simple 8-bit RISC machine with a Harvard architecture, 32 general-purpose registers, an 8-bit integer multiplier, and several peripherals. Utilizing the integer multiplier efficiently is crucial to achieve high performance in prime-field arithmetic. In the next two sections, we describe and evaluate ECC software implementations on three different AVR platforms. The first is a standard ATmega128 (or any other AVR processor that has the same instruction timing, e.g. our JAAVR in cycle-accurate mode as discussed below). Our second platform is JAAVR operating in a mode with better CPI (i.e. less instruction cycles) than the original ATmega128. Finally, the third platform is JAAVR featuring a special (32×4)-bit MAC unit and other extensions.

JAAVR (our ATmega128 clone) is written in VHDL, can be fully synthesized to ASIC and FPGA technologies, and is fully instruction-set compatible with the original ATmega128 core. It supports two operation modi, which can be selected via a "generic" statement in the VHDL code [25]. When the flag `CYCLE_ACCURACY` is switched on, JAAVR has exactly the same CPI (cycles per instruction) timing as the original ATmega128 (except minor differences that are irrelevant for this paper). In this mode, JAAVR automatically inserts `NOP`s to be compatible with the ATmega128. Such a cycle-accurate mode is especially important for real-time applications (e.g. protocol handling) and makes it easy for a designer to switch from an original ATmega128 to our JAAVR. On the other hand, when `CYCLE_ACCURACY` is switched off, the CPI-count of most load (resp. store) and multiply instructions improves, which boosts the performance of all four ECC implementations discussed in this paper. Of course, not only ECC but many other kinds of IoT applications profit from the better CPI. However, the performance gain is especially pronounced for ECC since the underlying prime-field arithmetic requires to execute a large number of load, store, and multiply instructions [10]. In the following subsection, we extend the architecture of JAAVR to further reduce the execution time of ECC.

A. (32×4)-bit Multiply-Accumulate Unit

The performance of ECC relies on the efficiency of certain arithmetic operations in the underlying finite field [10]. The by far most important field operations are multiplication and squaring; optimizing these operations can drastically improve the execution time of an ECC implementation.

In order to speed up multiplication in $\mathbb{F}_p$, we enhanced the JAAVR core with a (32×4)-bit Multiply-Accumulate (MAC) unit that we tightly integrated into the processor core. A full (32×32)-bit multiplication yielding a 64-bit result has to be composed of eight (32×4)-bit MAC operations, which take

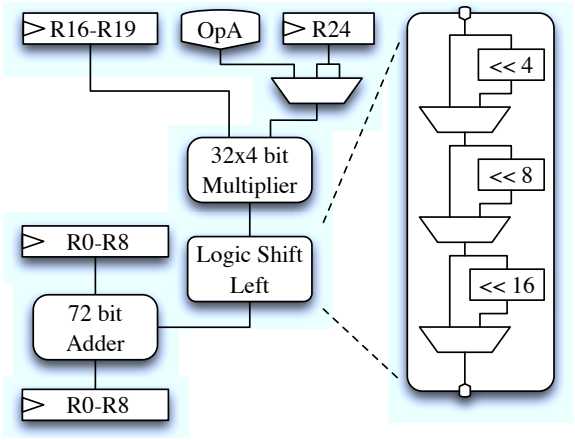

Fig. 1. (32×4)-bit Multiply-Accumulate unit.

eight clock cycles in total. However, the MAC does not stall the pipeline of the integer unit so that e.g. load instructions can be executed at the same time. Due to the CPI improvements mentioned above, JAAVR only needs a single cycle to execute a load instruction. Thus, two 32-bit operands (eight bytes) can be loaded during a (32×32)-bit multiplication. In other words, while performing a (32×32)-bit multiplication or MAC operation, JAAVR can already load the eight bytes of the two operands needed for the subsequent (32×32)-bit multiplication or MAC operation.

Figure 1 depicts a block diagram of our (32×4)-bit MAC unit. The first operand (a 32-bit word) is always read from registers `R16`–`R19`. As second operand, either the lower 4 bits of the selected operand register (OpA) or, alternatively, the lower or higher 4 bits of register `R24` is used. Since such a 4-bit nibble is only a part of the full 32-bit multiplicand, the 36-bit product has to be shifted by 0, 4, 8, 12, 16, 20, 24, or 28 bits. This is achieved with a barrel shifter, shown on the right of Figure 1. Using a 72-bit adder, the shifted product is accumulated into the fixed registers `R0`–`R8`. All this happens in a single clock cycle. The critical path of the JAAVR core is minimally affected by the integration of the MAC unit. For example, when synthesized using 130-nm standard cells, the maximum clock frequency of the JAAVR core is well above our desired operating frequency of 20 MHz[1].

The (32×4)-bit MAC unit is in principle suitable to speed up any public-key cryptosystem that relies on multi-precision multiplication, e.g. ECC over prime fields or even RSA. In this paper, we use OPFs [26] and perform the multiplication modulo $p = u \cdot 2^k + 1$ via a special variant of the FIPS method for Montgomery multiplication [14]. As described in Section III, our baseline implementation of OPF arithmetic operates on 32-bit words (i.e. 4 bytes are processed at a time), which makes it relatively straightforward to use the MAC unit. In order to minimize the execution time and fully exploit the MAC unit, we modified the baseline OPF implementation as follows. First, we completely unrolled the nested loops of the FIPS method to eliminate all loop overhead. Of course, loop unrolling bloats the code size, but using our MAC unit limits this effect since a single MAC operation replaces a multitude of AVR instructions. Therefore, unrolling the FIPS method is a viable optimization technique on JAAVR, but not necessarily on an ordinary AVR processor. Second, we used `Y` and `Z` as registers to access the two operands of an OPF multiplication since they allow one to execute a load instruction with a fixed offset (e.g. `LDD Rd, Y+offset`) so that neither `Y` nor `Z` are modified throughout a multiplication. Third, we replaced the (unrolled) assembly code for the (32×32)-bit multiplications by a sequence of eight MAC operations.

[1]Note that conventional AVR-based sensor nodes, such as the widely used MICAz motes, are clocked with a frequency of only 7.3728 MHz.

Algorithm 1. 32-bit MAC operation reusing `SWAP` instruction

```
 1: LD R16, Y+0        9: SWAP R20
 2: LD R17, Y+1       10: SWAP R20
 3: LD R18, Y+2       11: SWAP R21
 4: LD R19, Y+3       12: SWAP R21
 5: LD R20, Z+0       13: SWAP R22
 6: LD R21, Z+1       14: SWAP R22
 7: LD R22, Z+2       15: SWAP R23
 8: LD R23, Z+3       16: SWAP R23
```

We implemented two alternative mechanisms to access the MAC unit; Algorithm 1 illustrates one of these. In a special mode of operation, configured using an I/O-memory-mapped register, the `SWAP` instruction gets "re-interpreted." During the execution of `SWAP`, the lower and higher 4 bits of the chosen register are exchanged, and additionally the lower four bits are multiplied by the 32-bit word stored in registers `R16` to `R19`. Thus, eight `SWAP`s are necessary for a (32×32)-bit multiplication. The offset for the "Logic Shift Left" (Figure 1) is automatically generated by an internal counter that overflows after eight multiplications, resetting itself to zero.

Algorithm 2 shows a second way (with better performance) to accomplish the MAC operation. For every load operation (`LDD`, `LD`) with `R24` being used as the destination register, two MAC operations are performed automatically during the two following clock cycles. Since the MAC operations are carried out on the MAC unit, the ALU is free and can execute some other instructions in parallel, provided that these instructions do not access any of the 13 accumulator (resp. multiplicand) registers. The `NOP` instructions shown in Algorithm 2 can be replaced by operations to load operands for the subsequently performed (32×32)-bit multiplication.

Our carefully optimized Assembly code for multiplication in a 160-bit OPF requires exactly 552 cycles. This execution time can be broken down into 204 load instructions (`LD` and `LDD`), of which 100 activate a MAC operation, 40 store (`ST`) instructions, 83 `MOVW` instructions, 40 `SWAP` instructions, and 31 `NOP` instructions, which are actually only needed because of data-dependencies.

V. Results and Discussion

We describe and discuss results in three categories: (1) the impact of the hardware optimizations on the execution time

Algorithm 2. 32-bit MAC operation: $L()$ and $H()$ are used to access the lower and higher 4 bits of the loaded word

```
 1: LDD R16, Y+0
 2: LDD R17, Y+1
 3: LDD R18, Y+2
 4: LDD R19, Y+3
 5: LDD R24, Z+0
 6: NOP              (acc ← acc + (A × L(Z+0)) ≪ 0)
 7: LDD R24, Z+1     (acc ← acc + (A × H(Z+0)) ≪ 4)
 8: NOP              (acc ← acc + (A × L(Z+1)) ≪ 8)
 9: LDD R24, Z+2     (acc ← acc + (A × H(Z+1)) ≪ 12)
10: NOP              (acc ← acc + (A × L(Z+2)) ≪ 16)
11: LDD R24, Z+3     (acc ← acc + (A × H(Z+2)) ≪ 20)
12: NOP              (acc ← acc + (A × L(Z+3)) ≪ 24)
13: NOP              (acc ← acc + (A × H(Z+3)) ≪ 28)
```

TABLE I
OPF OPERATIONS ON DIFFERENT VARIANTS OF JAAVR.

Processor Mode	CA	FAST	ISE
Runtimes [Cycles]			
Addition	240	145	145
Subtraction	240	145	145
Multiplication	3,314	2,537	552
Inversion	189k	128k	124k
Chip Area			
JAAVR	6,166 GE	6,800 GE	8,344 GE
Difference	–	+10%	+35%

of OPF arithmetic, (2) the ATmega128-compatible execution time of the different point multiplication methods, and (3) the area requirements of the different variants of JAAVR.

A. Arithmetic Operations in OPFs

As mentioned previously, the execution time of arithmetic operations in the underlying field has a direct impact on the performance of the point multiplication. Table I summarizes three types of execution times for arithmetic operations in a 160-bit OPF: CA, FAST, and ISE. CA is the cycle-accurate mode of our JAAVR, which has exactly the same instruction timing as a standard ATmega128. When the cycle-accuracy is switched off, JAAVR operates in FAST mode. The ISE mode adds the previously described MAC unit to JAAVR.

When unrolling the loop, addition (and also subtraction) in a 160-bit OPF takes 240 cycles on JAAVR in CA mode. The runtime of addition (resp. subtraction) is constant (i.e. does not depend on the operands) with a probability of $1-2^{-32}$ as explained in Section III. By improving the `LD`, `ST`, `PUSH`, and `POP` instructions, utilizing the FAST (i.e. non cycle-accurate) mode of JAAVR, the runtime improved to 145 cycles, which corresponds to a speed-up factor of 1.65. Unfortunately, the impact of load and store optimizations on the multiplication time is less significant. A "looped" OPF multiplication needs 3,314 cycles in CA versus 2,537 cycles in FAST mode, i.e. a speed-up factor of 1.31 was achieved. However, reducing the execution cycles of instructions does not come for free. Some 634 GE (10 %) of logic have to be added to the CA version of JAAVR, which occupies roughly 6.2 kGE. Fortunately, the speed-up is larger than the increase in area, which means the area-time product got improved. Further 1.5 kGE (+23 %) are necessary to integrate the proposed MAC unit. It reduces the cycle-count for a 160-bit OPF multiplication to 552, which is 4.6 (resp. 6.0) times faster than the software implementation in FAST (resp. CA) mode. The impact of our extensions on the performance of point multiplication is discussed below.

TABLE II
POINT MULTIPLICATION TIMES ON A STANDARD ATMEGA128.

Elliptic Curve	High-speed Method	High-speed Runtime	Constant Round Method	Constant Round Runtime
		[kCycles]		[kCycles]
Standardized Curve				
`secp160r1`	NAF	7,136	Mon	8,722
Non-Standardized Curves using OPF				
Weierstraß	NAF	6,983	Mon	8,824
Edwards	NAF	5,597	DAAA	8,251
Montgomery	Mon	5,545	Mon	**5,545**
GLV	End, JSF	**3,930**	Mon	8,132

B. Elliptic Curve Point Multiplications

Table II shows the results of all implemented curve shapes on a standard ATmega128 (or JAAVR in CA mode). We used different point multiplication methods to achieve either high speed or reduced side-channel leakage via "branch-less" code with constant (i.e. operand-independent) execution time. The implementations belonging to the latter category do not have fully constant execution time due to the Montgomery inverse we used in projective-to-affine conversion [10], but the main loop of the algorithm for point multiplication is performed in constant time, independent of the value of the scalar.

The standardized curve `secp160r1` does not use an OPF as underlying field; therefore, its field arithmetic is implemented with a separate set of assembly-optimized functions. For the field multiplication, an unrolled variant of Gura et al's hybrid multiplication method [9] is used in combination with some prime-specific optimizations of the modular reduction.

We decided to stick with methods for point multiplication that require a minimal amount of memory. In general, ECC is only part of a larger application (e.g. a security protocol) and should, therefore, not consume all available program or data memory. Also, no comb methods with pre-calculated points are used since they require the base point P to be fixed and known a-priori (i.e. a comb method is not suitable to perform ECDH key exchange). The double-and-add algorithm based on Non-Adjacent Form (NAF) representation of the scalar is well suited for fast ECC with low memory footprint because it reduces the number of point additions but does not demand much extra memory [10]. In combination with an appropriate coordinate systems (specified in brackets), the NAF method was used for `secp160r1` (Jacobian [10]), Weierstraß (same Jacobian [10]), and Edwards curves (Hişil et al [12]). For the GLV curve, the Jacobian doubling formula was optimized to save a multiplication since $a = 0$. The GLV implementation exploits the endomorphism mentioned in Section II and uses a Joint Sparse Form (JSF) representation for k_1, k_2 [10].

To get constant (i.e. scalar-independent) execution time, we employed the Montgomery ladder [3] and Double-And-Add-Always (DAAA) algorithm. For `secp160r1`, Weierstraß, and GLV curves, the 10-register formula by Hutter et al [13] was applied. There exist no ladder-like (i.e. differential) addition formulas for Edwards curves, but due to completeness of the addition law, the DAAA algorithm can be implemented in a fairly straightforward way. The Montgomery curve is special because it features differential point addition formulas of high regularity and low complexity [3]. In fact, these formulas are so efficient that the performance-optimized and constant-time implementations are the same. The GLV curve achieved the fastest point multiplication time, requiring only 3,930 kCycles on an ATmega128. Montgomery, Edwards, Weierstraß, and `secp160r1` elliptic curves are about 41 %, 42 %, 77 %, and 82 % slower, respectively. The fastest of the leakage-reduced implementation is the Montgomery curve with 5,945 kCycles (41 % slower). All other "low-leakage" implementations are roughly equal, ranging between 8.2 and 8.8 MCycles.

C. Architectural Support for OPF-Based ECC

While the previous performance figures were obtained on a standard ATmega128 (or JAAVR in CA mode), we now take advantage of the FAST and ISE modes. Table III reports the execution time and synthesis results (i.e. silicon area) of all performed experiments. Our target technology was a 130 nm low-leakage CMOS process from UMC with Faraday design library. The silicon area was minimized by utilizing area-optimized, synchronous one-port register-file RAM macros. Thus the impact of the required data memories (528, 505, 567, and 865 bytes for Weierstraß, Montgomery, Edwards, and GLV curves, respectively) on the total chip area is rather small.

The most complex point multiplication algorithm requiring the biggest program memory is the GLV method. Its program memory is approximately 43 % larger than that needed by the Edwards curve, which is with some 19.6 kGE altogether the smallest of all our implementations. Switching from CA to FAST and from FAST to ISE increases the silicon area of all implementations, but also improves their execution time. This is best illustrated by the results for the Scaled Area-Runtime Product (SARP) from Table III. In CA and FAST mode, the GLV curve achieves the highest SARP value and, hence, has the best area-runtime product. However, if in ISE mode, the Edwards curve outperforms the other three implementations by a small margin (SARP between 5.06 and 5.13).

By utilizing the MAC unit, the OPF-multiplication becomes roughly six times faster, but the overall point multiplications only improved by factors of between 3.9 and 4.5. This is to be expected since the MAC unit accelerates only the multiplication but no other OPF operations. Switching from CA to FAST mode improved the execution times by some 33 % and all area-runtime products by 28 %, respectively. Table III also shows simulated power consumption values for a placed and routed design at 1 MHz. While the CPU (17–22 μW) and the RAM (1.2–5.4 μW) require relatively low power, the program memory (which was synthesized using logic cells) consumes up to 110 μW. Of course, replacing this synthesized program memory by a "Via-1" ROM macro would reduce the overall power consumption significantly. The energy cost of a single point multiplication on JAAVR in CA mode ranges between 455 μJ (GLV curve) and 969 μJ (Weierstraß curve).

TABLE IV
COMPARISON WITH RELATED HARDWARE IMPLEMENTATIONS.

Reference	Field Type	Field Size	Runtime [kCycles]	Area [GE]
Koschuch et al. [15]	$GF(2^m)$	163 bit	1,190	29,491
Fürbass et al. [5]	$GF(p)$	160 bit	362	19,000
Hein et al. [11]	$GF(2^m)$	163 bit	**296**	13,250
Lee et al. [16]	$GF(2^m)$	163 bit	302	12,506
Wenger et al. [25]	$GF(p)$	192 bit	1,377	**11,686**
Our Work (Mon)	$GF(p)$	160 bit	1,300	20,980

TABLE V
RELATED SOFTWARE IMPLEMENTATIONS ON AN ATMEGA128.

Reference	Elliptic Curve	Runtime [kCycles]
Liu and Ning [17]	`secp160r1`	15,060
Wang and Li [23]	`secp160r1`	9,953
Szczechowiak et al. [21]	Weierstraß, GM prime	9,376
Ugus et al. [22]	`secp160r1`	7,594
Gura et al. [9]	`secp160r1`	6,480
Großschädl et al. [8]	GLV, OPF	5,480
Our Work	Montgomery, OPF	5,545
Our Work	GLV, OPF	**3,930**

D. Comparison with Related Work

We distinguish two categories of related work: (1) "light-weight" hardware implementations of ECC, and (2) software implementations of ECC for the ATmega128 based on prime fields. Only very few hardware/software co-designs exist; we include them in the former category in our comparison. Table IV summarizes dedicated hardware implementations of ECC for resource-restricted environments. Although most of them outperform our ASIP in terms of runtime and area, they are less flexible and scalable than our co-design because they can not handle different fields or families of curve. Our ASIP, on the other hand, includes a C programmable AVR core able to perform various other tasks, e.g. protocol processing.

Table V shows a summary of related ATmega128 software implementations. Our "pure" software solutions (using native AVR instructions only) outperform most previously-reported software implementations of ECC over prime fields.

VI. CONCLUSIONS

We presented an AVR-compatible 8-bit ASIP for ECC and evaluated its performance through software implementations of several scalar multiplication algorithms. The research contribution of this paper is twofold. First, we showed that the integration of a (32×4)-bit MAC unit into an AVR core is a viable option to speed up ECC since it allows one to achieve a threefold performance gain at the cost of a slight increase in area. Second, we analyzed the efficiency of four families of elliptic curves on our ASIP, taking into account numerous implementation options such as the use of the MAC unit and whether or not the scalar multiplication method should have

TABLE III
SYNTHESIS RESULTS FOR THE DIFFERENT MODES OF JAAVR (HIGHER SARP VALUE MEANS BETTER AREA-RUNTIME PRODUCT).

Elliptic Curve	Mode	Point Mult. [Cycles]	ROM [bytes]	JAAVR [GE]	ROM [GE]	RAM [GE]	Total [GE]	JAAVR [μW]	ROM [μW]	Total [μW]	SARP
Weierstraß	CA	6,982,629	6,224	6,166	9,091	4,485	19,742	18.8	109.5	138.8	1.00
Edwards	CA	5,596,860	6,022	6,166	8,694	4,712	**19,572**	18.0	81.9	110.1	1.26
Montgomery	CA	5,545,078	6,824	6,167	9,542	4,359	20,068	17.9	60.0	88.9	1.24
GLV	CA	3,930,256	8,638	6,166	12,413	6,450	25,029	16.8	87.1	115.7	1.40
Weierstraß	FAST	5,254,706	6,224	6,800	9,071	4,485	20,355	18.6	60.2	89.7	1.29
Edwards	FAST	4,214,289	6,022	6,802	8,695	4,712	20,208	19.4	50.1	80.9	1.62
Montgomery	FAST	4,165,405	6,824	6,803	9,533	4,359	20,695	18.3	15.4	**45.4**	1.60
GLV	FAST	2,939,929	8,638	6,802	12,413	6,450	25,665	19.5	68.0	99.9	1.83
Weierstraß	ISE	1,542,981	6,290	8,344	8,718	4,485	21,546	18.7	58.4	88.5	4.15
Edwards	ISE	1,230,663	6,128	8,345	8,562	4,359	21,266	20.7	67.3	99.8	**5.27**
Montgomery	ISE	1,299,598	**5,752**	8,343	7,926	4,712	20,980	21.8	14.4	49.5	5.06
GLV	ISE	**1,001,302**	8,640	8,330	12,078	6,450	26,858	19.5	78.5	111.1	5.13

a regular execution profile to counteract certain side-channel attacks. Based on our experimental results, we can draw the following conclusions: The GLV curve is an excellent option when high speed is the main criterion because it outperforms all other curve types by at least 23%. On the other hand, if a regular execution profile (and operand-independent execution time) is important, the Montgomery curve is clearly the best choice. Last but not least, the Edwards curve has its benefits when the area-time product is of primary importance.

ACKNOWLEDGMENTS

The research described in this paper has been supported, in part, by the European Commission through the ICT Program under contract ICT-SEC-2009-5-258754 TAMPRES.

REFERENCES

[1] Atmel Corporation. 8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash. Datasheet, available online at http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf, June 2011.

[2] D. J. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In *Advances in Cryptology — ASIACRYPT 2007*, LNCS 4833, pp. 29–50. Springer Verlag, 2007.

[3] H. Cohen and G. Frey (eds). *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, vol. 34 of Discrete Mathematics and Its Applications. Chapmann & Hall\CRC, 2006.

[4] H. M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44(3):393–422, July 2007.

[5] F. Fürbass and J. Wolkerstorfer. ECC processor with low die size for RFID applications. In *Proceedings of the 40th IEEE International Symposium on Circuits and Systems (ISCAS 2007)*, pp. 1835–1838. IEEE, 2007.

[6] R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphism. In *Advances in Cryptology — CRYPTO 2001*, LNCS 2139, pp. 190–200. Springer Verlag, 2001.

[7] J. Großschädl. TinySA: A security architecture for wireless sensor networks. In *Proceedings of the 2nd International Conference on Emerging Networking Experiments and Technologies (CoNEXT 2006)*, pp. 288–289. ACM Press, 2006.

[8] J. Großschädl, M. Hudler, M. Koschuch, M. Krüger, and A. Szekely. Smart elliptic curve cryptography for smart dust. In *Quality of Service in Heterogeneous Networks — QSHINE 2010*, LNICST 74, pp. 548–559. Springer Verlag, 2010.

[9] N. Gura, A. Patel, A. S. Wander, H. Eberle, and S. Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems — CHES 2004*, LNCS 3156, pp. 119–132. Springer Verlag, 2004.

[10] D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.

[11] D. Hein, J. Wolkerstorfer, and N. Felber. ECC is ready for RFID — A proof in silicon. In *Selected Areas in Cryptography — SAC 2008*, LNCS 5381, pp. 401–413. Springer Verlag, 2009.

[12] H. Hişil, K. K.-H. Wong, G. Carter, and E. Dawson. Twisted Edwards curves revisited. In *Advances in Cryptology — ASIACRYPT 2008*, LNCS 5350, pp. 326–343. Springer Verlag, 2008.

[13] M. Hutter, M. Joye, and Y. Sierra. Memory-constrained implementations of elliptic curve cryptography in co-Z coordinate representation. In *Progress in Cryptology — AFRICACRYPT 2011*, LNCS 6737, pp. 170–187. Springer Verlag, 2011.

[14] Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[15] M. Koschuch, J. Lechner, A. Weitzer, J. Großschädl, A. Szekely, S. Tillich, and J. Wolkerstorfer. Hardware/software co-design of elliptic curve cryptography on an 8051 microcontroller. In *Cryptographic Hardware and Embedded Systems — CHES 2006*, LNCS 4249, pp. 430–444. Springer Verlag, 2006.

[16] Y. K. Lee, L. Batina, K. Sakiyama, and I. Verbauwhede. Elliptic curve based security processor for RFID. *IEEE Transactions on Computers*, 57(11):1514–1527, Nov. 2008.

[17] A. Liu and P. Ning. TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN 2008)*, pp. 245–256. IEEE Computer Society, 2008.

[18] C. P. Mayer. Security and privacy challenges in the Internet of things. *Electronic Communications of the EASST*, 17(4):1–12, Mar. 2009.

[19] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.

[20] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, Jan. 1987.

[21] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab. NanoECC: Testing the limits of elliptic curve cryptography in sensor networks. In *Wireless Sensor Networks — EWSN 2008*, LNCS 4913, pp. 305–320. Springer Verlag, 2008.

[22] O. Ugus, D. Westhoff, R. Laue, A. Shoufan, and S. A. Huss. Optimized implementation of elliptic curve based additive homomorphic encryption for wireless sensor networks. In *Proceedings of the 2nd Workshop on Embedded Systems Security (WESS 2007)*, pp. 11–16, 2007.

[23] H. Wang and Q. Li. Efficient implementation of public key cryptosystems on mote sensors. In *Information and Communications Security — ICICS 2006*, LNCS 4307, pp. 519–528. Springer Verlag, 2006.

[24] E. Wenger, T. Baier, and J. Feichtner. JAAVR: Introducing the next generation of security-enabled RFID tags. In *Proceedings of the 15th EUROMICRO Conference on Digital System Design (DSD 2012)*, pp. 640–647. IEEE Computer Society, 2012.

[25] E. Wenger, M. Feldhofer, and N. Felber. Low-resource hardware design of an elliptic curve processor for contactless devices. In *Information Security Applications — WISA 2010*, LNCS 6513, pp. 92–106. Springer Verlag, 2011.

[26] Y. Zhang and J. Großschädl. Efficient prime-field arithmetic for elliptic curve cryptography on wireless sensor nodes. In *Proceedings of the 1st International Conference on Computer Science and Network Technology (ICCSNT 2011)*, vol. 1, pp. 459–466. IEEE, 2011.