Architectural Extensions for Elliptic Curve Cryptography over GF(2^m)

Hans Eberle, Arvinderpal Wander, Nils Gura, Sheueling Chang-Shantz
Sun Microsystems Laboratories
2600 Casey Aveneue
Mountain View, CA 94043

Abstract

We describe data path extensions for general-purpose microprocessors to accelerate the emerging public-key cryptosystem Elliptic Curve Cryptography (ECC). ECC is computationally more efficient than the popular RSA cryptosystem and, thus, is an enabling security technology for light-weight devices that are limited in compute power, memory capacity, and battery power.

Elliptic curves have been standardized by NIST and SECG for fields $GF(p)$ and $GF(2^m)$. Though both types of fields offer similar security strengths, the standards offer a choice to accommodate different implementation platforms. While arithmetic operations over fields $GF(p)$ directly map to integer operations found in standard processors, operations over fields $GF(2^m)$ are supported rather inefficiently.

We show that simple extensions of the data path suffice to efficiently support ECC over $GF(2^m)$ and to outperform ECC over $GF(p)$. These extensions include an extended integer multiplier that also generates multiplication results for fields $GF(2^m)$ and a multiply-accumulate instruction for efficient multiple-precision multiplications.

On the 8-bit ATmega128 microprocessor running at 8 MHz we measured an execution time for a 163-bit ECC point multiplication over $GF(2^m)$ of 0.4 s with the extended multiplier and 0.29 s if, in addition, a multiply-accumulate instruction is provided. In comparison, a 1024-bit RSA private-key operation providing equivalent security strength takes 11 s.

1 Introduction

In this paper, we examine hardware extensions at the microarchitecture level to accelerate public-key operations with elliptic curve cryptography (ECC). While the extensions proposed are generic in that they can be applied to any general purpose processor, this paper addresses their implementation on 8-bit microprocessors.

Typically, acceleration of public-key cryptosystems is achieved through dedicated coprocessors. Here, we want to explore an alternative approach that provides hardware acceleration through extensions of an existing data path of a general-purpose processor. This approach is a more attractive option as it requires fewer chip resources, in particular, if logic can be shared by the processor core and the extensions, and as it further provides a seamless programming interface.

Due to its computational efficiency, ECC is emerging as an attractive alternative to traditional public-key cryptosystems such as RSA, DSA, and DH [19]. More specifically, ECC offers equivalent security with smaller key sizes, resulting in reduced computation time and lower memory requirements. Thus, ECC is particularly well suited for mobile and wireless applications running on light-weight devices that are typically constrained in the amount of available compute power, memory capacity, and battery power. In this paper, we focus on the low end of the spectrum of devices, namely those based on 8-bit microprocessors. Examples of applications of 8-bit devices requiring public-key cryptographic operations are smart cards, sensor networks, telemetry, and home automation.

For the implementations described in this paper, we have chosen the ATmega128 8-bit AVR microprocessor from Atmel Inc. that employs a modern microprocessor architecture based on a reduced instruction set. This microprocessor is at the heart of the Motes devices originally developed at UC Berkeley [15, 12], which are a popular platform for research in sensor networks and that could benefit from support for public-key cryptography.

2 RSA and ECC Operations

The RSA cryptosystem is based on *modular exponentiation* which can be implemented through repeated multiplication and squaring. The security of RSA relies on the difficulty of factoring large integer values. The fundamental operation provided by the ECC cryptosystem is called *point multiplication*. Here, the underlying hard problem is known as the elliptic curve discrete logarithm problem (ECDLP). While sub-exponential algorithms exist for factoring large numbers, only exponential algorithms are known for the ECDLP. It is for this reason, that RSA requires larger key sizes than ECC to provide equivalent security strengths.

To illustrate the key size advantage of ECC, a 163-bit ECC key offers the same security strength as a 1024-bit RSA key. The ratio of key sizes is going to favor ECC even more as larger keys are adopted. To exemplify this, a 571-bit ECC key is comparable in security strength to a 15,360-bit RSA key.

Both, the RSA modular exponentiation and the ECC point multiplication are based on modular arithmetic over finite fields. RSA uses integer rings whereas ECC is defined over prime integer fields $GF(p)$ and binary polynomial fields $GF(2^m)$. Arithmetic operations in integer rings and over fields $GF(p)$ can be easily implemented with the standard integer operations available on a general-purpose processor. This is not the case for arithmetic operations over fields $GF(2^m)$. Though these operations could be implemented in hardware rather efficiently, executing them on standard processors is prohibitively slow. For this reason, software implementations of the ECC cryptosystem on standard processors resort to curves over $GF(p)$.

In this paper, we propose simple extensions to an existing data path of a general-purpose microprocessor that provide hardware support for operations over $GF(2^m)$.

3 Arithmetic in GF(2^m)

In this section, we want to briefly introduce the arithmetic operations needed to implement ECC point multiplication over binary polynomial fields $GF(2^m)$[1]. The opera-

[1]For an in-depth discussion of the math underlaying ECC, the reader is referred to [11]

tions include modular addition, subtraction, multiplication, squaring, and division, where the operands are polynomials with coefficients of either 0 or 1. We use the polynomial basis representation where a polynomial $a(t) \in GF(2^m)$ in canonical form is written as $a(t) = a_{m-1}t^{m-1} + a_{m-2}t^{m-2} + \cdots + a_1 t + a_0, a_i \in GF(2)$. For computation purposes, an $m-$bit binary vector can be used to represent the coefficients. For example, the polynomial $t^4 + t^3 + 1$ can be written as 11001.

3.1 Addition

The addition of two elements $a(t), b(t) \in GF(2^m)$ is computed by adding the coefficients a_i and b_i modulo 2, which corresponds to a bit-wise XOR operation:

$$a(t) + b(t) = \sum_{i=0}^{m-1} ((a_i + b_i)\, mod\, 2) * t^i = \sum_{i=0}^{m-1} (a_i \oplus b_i) * t^i$$

For example, a polynomial addition $(t^4 + t^2 + 1) + (t^3 + t^2 + t) = t^4 + t^3 + t + 1$ can be computed as $10101 \oplus 01110 = 11011$. Since every element of $GF(2^m)$ is identical to its additive inverse, subtraction is identical to addition.

3.2 Multiplication

Multiplication of two elements $a(t), b(t) \in GF(2^m)$ is carried out in two steps. First, the operands are multiplied using polynomial multiplication resulting in

$$\begin{aligned} c = a * b = & c_{2(m-1)} t^{2(m-1)} \\ & + c_{2(m-1)-1} t^{2(m-1)-1} \\ & + \cdots + c_1 t + c_0 \end{aligned}$$

The degree of c is less than $2m-1$, i.e. $\deg(c) < 2m-1$. The coefficients of c are calculated through convolution of a and b

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} \tag{1}$$

c may not be in reduced canonical form since its degree may be greater than $m-1$. Since the summation of the partial products is carried out by XOR operations, we refer to multiplications over $GF(2^m)$ as XOR multiplications. In a second step, c is reduced by an irreducible polynomial M. M is of degree m and defines $GF(2^m)$ for a chosen field degree m. The reduced canonical result $r \equiv c \mod M$, $\deg(r) < m$ is defined as the residue of the polynomial division of c by M. For example, given polynomials $a = t^3 + t + 1$ and $b = t^3 + 1$ of $GF(2^4)$, represented as $a = 1011$ and $b = 1001$, $c = a * b = t^6 + t^4 + t + 1$ can be computed as shown in Figure 1. Assuming $M = t^4 + t^3 + 1$, represented as $M = 11001$, the reduction $r = c \mod M = t^2 + 1$ can be performed as shown in Figure 2. An illustrative way to look at reduction is that M is aligned with the most significant bit of the operand and added until the degree of the result is smaller than m.

```
1011  *   1001 (t^3 + t + 1) * (t^3 + 1)
--------------
          1001
xor      1001
xor     0000
xor    1001
--------------
 =     1010011 (t^6 + t^4 + t + 1)
```

Figure 1. *Polynomial multiplication.*

```
    1010011 (t^6 + t^4 + t + 1)
xor 11001   (t^6 + t^5 + t^2)
-----------
    0110111 (t^5 + t^4 + t^2 + t + 1)
xor  11001  (t^5 + t^4 + t^1)
-----------
 =  0000101 (t^2 + 1)
```

Figure 2. *Polynomial reduction.*

The implementation described in this paper uses optimized reduction made possible by the pentanomials and trinomials chosen as irreducible polynomials for the curves standardized by NIST and SECG [2, 3]. This technique yields significantly faster reduction than other generic reduction techniques [5, 4]. Optimized reduction is attractive for light-weight devices as they have to support only a small number of elliptic curves.

3.3 Squaring

Polynomial squaring of an element $a(t) \in GF(2^m)$ results in

$$a(t)^2 = \sum_{i=0}^{m-1} a_i * t^{2i} = a_{m-1}t^{2m-2} + \cdots + a_2t^4 + a_1t^2 + a_0$$

The binary representation of $a(t)^2$ is obtained by inserting 0s between the bits of the original vector representing $a(t)$. For example, squaring $t^4 + t^3 + 1$ yields, in binary representation, 101000001.

3.4 Inversion

The computation of an inversion in $GF(2^m)$ is computionally expensive and takes considerably more execution time than addition or multiplication. There are two types of algorithms used to implement inversion: Fermat's little theorem $a^{p-1} \equiv 1 \mod p$ applied to $GF(2^m)$ and Euclid's greatest common divisor algorithm. We use a variant of the latter described by Chang Shantz in [1].

Fortunately, there are algorithmic choices that influence the ratio of multiplications and inversions. For the implementations described in this paper, we use projective coordinates that reduce the number of inversions needed to compute a point multiplication to just one.

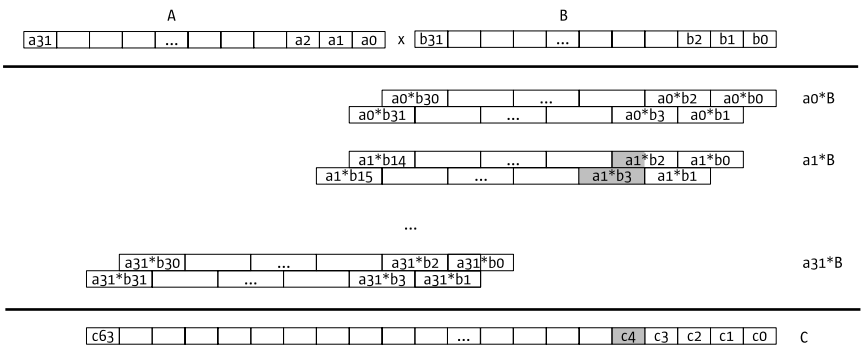

Figure 3. Multiple-precision multiplication.

4 Multiple-precision Multiplication

Multiple-precision multiplication is the most time-critical operation underlying both the RSA and the ECC cryptosystems. Figure 3 depicts the calculation of a multiple-precision multiplication and how it is broken up into multiplication instructions. The given example assumes 256-bit operands A and B. Operands are broken up into 8-bit words $(a_{n-1}, .., a_0)$ and $(b_{n-1}, .., b_0)$, $n = 32$. The product $C = \sum_{i=0}^{n-1}(a_i * B * t^{8i})$ is the sum of the partial products $a_i * B$. To calculate a partial product, $n = 32$ 8x8 multiplications are required, and to calculate the final product, $n^2 = 1024$ 8x8 multiplication operations have to be executed. There is a choice in the order in which the multiplications are executed and the resulting products are summed up. The corresponding techniques are typically referred to as column-wise and row-wise multiplication methods. Gura *et al.* describe a hybrid technique in [6] that reduces memory accesses and, thus, yields higher performance. In their paper, the new technique is accredited with a performance gain of 25 % for ECC point multiplication over fields $GF(p)$ on the Atmel AVR platform. We have adopted this technique for the implementation described in this paper.

RSA and ECC are both based on modular arithmetic. Thus, the multiple-precison multiplication operations require an additional reduction step. For RSA, we apply the Montgomery technique [17] that replaces the costly division operation required to calculate the remainder of the modulo operation with a simple addition of a multiple of the modulus. For ECC, we make use of the fact that the moduli of the standardized curves represent sparse irreducible polynomials that make it possible to implement reduction by only a few shift and add operations.

5 Multiple-precision Squaring

Multiple-precision squaring over fields $GF(2^m)$ can be done in linear time $O(n)$. This is unlike squaring over fields $GF(p)$ which has complexity $O((n/2)^2)$. Figure 4 illustrates squaring of a 256-bit operand representing a polynomial of degree 256. The operand is split into 32 bytes $(a_{n-1}, .., a_0)$, $n = 32$. The square is obtained by multiplying each byte a_i with itself. The implementation described in the following does not use an optimized square instruction as the performance gain over a regular multiplication is minimal.

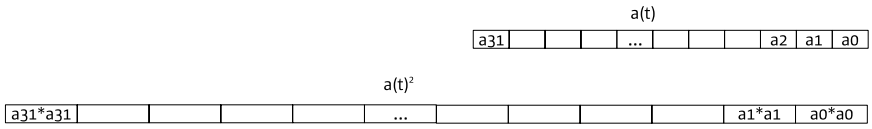

Figure 4. Multiple-precision squaring.

6 Architectural Extensions

In this section, we describe architectural extensions to support modular arithmetic over fields $GF(2^m)$. We begin with describing two new instructions that accelerate multiplications over $GF(2^m)$. Next, we present an extended version of a standard integer multiplier that provides the results for these new instructions.

The new instructions are compatible with the existing AVR instruction set. In particular, we use a format that allows for specifying two operands only. The new instructions are also compatible with the existing data path in that they only load two source operands and store two destination operands.

6.1 MULX Instruction

We first introduce the XOR multiplication instruction:

$$\begin{aligned} &MULX\ R_d, R_r : \\ &R1 : R0 \leftarrow (R_d \otimes R_r) \end{aligned} \tag{2}$$

Operator $\otimes$ refers to a multiplication over fields $GF(2^m)$. MULX inputs two source operands specified by R_d and R_r and outputs the XOR multiplication result into register pair $R_0 : R_1$.

6.2 MULACCX Instruction

The multiplier can be optimized for multiple-precision operations by implementing a *multiply-accumulate instruction with extended carry* instruction that combines a multiplication step and an accumulation step:

$$\begin{aligned} &MULACCX\ R_d, R_r : \\ &R_d \leftarrow ((R_r \otimes R_c) \oplus XC \oplus R_d)[7:0] \\ &XC \leftarrow ((R_r \otimes R_c) \oplus XC \oplus R_d)[15:8] \end{aligned}$$

Operator $\otimes$ refers to a multiplication over fields $GF(2^m)$ and operator $\oplus$ stands for an addition over fields $GF(2^m)$.

The instruction specifies two source registers R_r and R_d, a destination register R_d (identical with one of the source operands), an implicit architectural register R_c, and a non-architectural register XC. Register R_c has to be pre-loaded with a load instruction prior to

executing MULACCX[2]. XC is referred to as the extended carry; it is local to the multiplier and cannot be accessed externally.

The MULACCX instruction can be applied to the multiple-precision multiplication as depicted in Figure 3 as follows. Referring to the example highlighted by the shaded areas, a single multiply-accumulate instruction MULACCX $R_r = B_n, R_d \leftarrow c_{n+p}$ would calculate $c_{n+p} = (a_p \otimes b_n) \oplus (a_p \otimes b_{n-1}[15:8]) \oplus c_{n+p}$ whereby R_c holds a_p and XC holds $(a_p \otimes b_{n-1}[15:8])$. In the shown example, $p = 1$ and $n = 3$.

While our explanations in this section have concentrated on field operations over $GF(2^m)$, the same multiplier structure shown in Figure 6 can be used to implement the integer multiply-accumulate instruction MULACC.

As the extended carry register XC cannot be explicitly accessed, there need to be other ways to load and save it, for example, upon entry and exit of a subroutine call. To save XC, it suffices to execute MULACCX $R_d = 0, R_r = 0$ – this instruction leaves the saved value in R_d. Loading a value into XC can be accomplished with one MULACCX instruction and a few supporting standard instructions. We will not provide the details of this procedure as the supporting math exceeds the scope of this section.

6.3 XOR Multiplier

We have developed a dual-field multiplier that generates both integer multiplication results as well as non-reduced multiplication results needed for multiplications in $GF(2^m)$. The multiplier is based on an architecture typically found in today's general-puprose processors. A dual-field multiplier similar to ours has also been described in [13].

Integer multipliers typically use a carry-save adder (CSA) tree together with a carry-propagate adder (CPA). The CSA tree calculates the sum of the partial products in a redundant carry/sum representation and the CPA performs the final addition of the carry and sum bits. We modified the CSA tree such that it generates the XOR product in addition to the integer product. The former is needed for ECC operations over fields $GF(2^m)$ and the latter for RSA and ECC operations over fields $GF(p)$.

Figure 5 shows the organization of the multiplier with an extension to support the MULX instruction. Registers A and B are m bits in size and hold the multiplicand and the multiplier, respectively. The partial products $a_i * B, i = 0..m-1$ are inputs to the CSA tree. For integer multiplications, the CSA tree generates a carry vector and a sum vector, and stores these vectors in registers P_c and P_s, respectively. The sum of P_c and P_s is computed by the CPA and written into register pair $R1 : R0$. For XOR multiplications, the XOR result is generated by the CSA tree and written into register pair $R1 : R0$.

To implement the MULACCX instruction, registers R_c and XC need to be added to the multiplier circuit as shown in Figure 6. The contents of these registers are added to the XOR multiplication result simply by making them additional input terms of the CSA tree. The XOR result generated by the CSA tree is split into two halves in that the low byte is written to R_d and the high byte is stored in XC.

A CSA tree consists of full adder (FA) elements and half adder (HA) elements. In its simplest form, such a tree uses $2n$ chains each consisting of 1 to n FAs and HAs to sum up n partial products. There are techniques to reduce or compress the chain lengths thereby reducing the logic delay to obtain the carry/sum result. With these techniques the tree height is reduced from n to $log_{1.5}n$ [9, 20].

[2]We refer to it as R_c as its content remains constant throughout the computation of partial product.

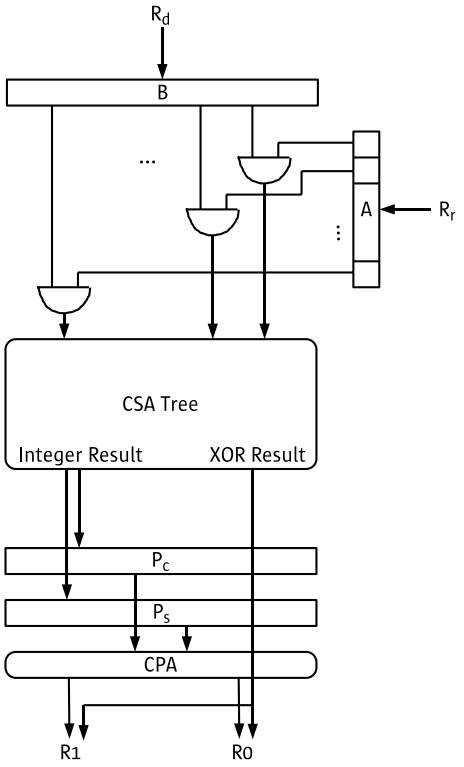

Figure 5. Multiplier with support for MULX.

We will now explain how the CSA tree can be modified to obtain the XOR result in addition to the integer result. Looking at the functions realized by the FAs and the HAs we notice that the sum S already provides the XOR function needed:

$$
\begin{aligned}
FA: \quad & S = A \oplus B \oplus C_{in} \\
& C_{out} = A \cdot B + A \cdot C_{in} + B \cdot C_{in} \\
HA: \quad & S = A \oplus B \\
& C_{out} = A \cdot B
\end{aligned}
$$

Thus, the XOR result can be obtained by chaining the FAs and the HAs in such a way that the inputs to the sum outputs are not connected to any carry output of a FA or HA. Figure 7a shows column 3 of a CSA tree for a 4x4 multiplier[3] and Figure 7b shows the modified version of the column that generates both the XOR result and the sum and the carry of the integer result.

The modifications shown require little extra circuitry - some columns require the addition of an XOR gate - and do not increase logic delays. Analyzing the extra cost of adding support for an additional XOR result to a multiplier, we found an average increase of 5% in terms of logic required. Not only are these modifications low in cost, they also do not increase the length of the critical path. Whereas previous designs of dual-field multipliers [7] have relied on a global signal that selects the field to be operated on, our design does

[3]The multiplier inputs six terms of which two are added to the product.

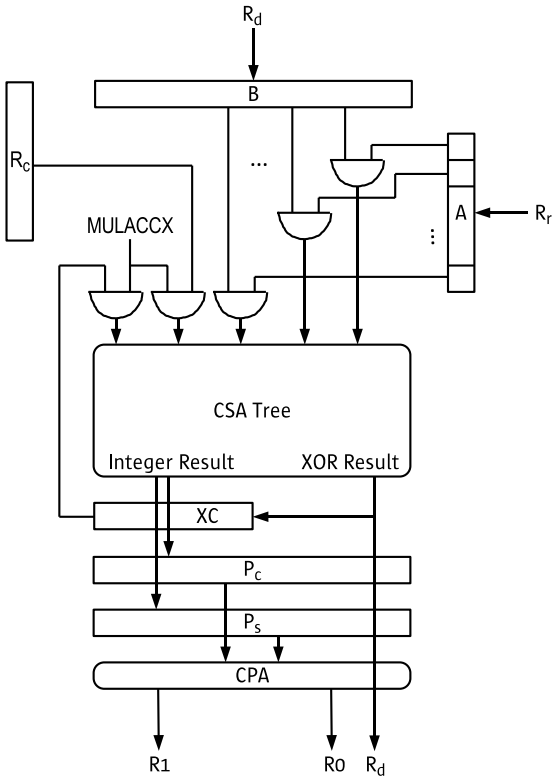

Figure 6. Multiplier with support for MULACCX.

not require any global communication that could easily become the critical path. Finally, it is worth pointing out that the outlined modifications can be easily applied to multiplier designs found in general-purpose CPUs.

7 Performance Evaluation

We have developed fully functional code for the Atmel ATmega128 8-bit microprocessor for ECC point multiplications over $GF(2^m)$. The code is written in assembly language to achieve optimal performance.

We give performance numbers for both RSA private-key and public-key operations. A message is encrypted by public-key operations and decrypted by private-key operations. In another scenario, the signature of a message is generated by a private-key operation and verified by a public-key operation. The RSA private-key operation is far more costly than the RSA public-key operation. We use a common technique known as the Chinese Remainder Theorem (CRT) to reduce the computation time for RSA private-key operations. With this technique, a modular exponentiation is split into two smaller exponentiation operations using operands for the base and exponents that are both half the size of the original operands. This reduces complexity from $O(n^3)$ to $O(2 * (n/2)^3)$, corresponding to a speedup factor of four. We chose not to use common performance optimizations such as window techniques or the Karatsuba Ofman multiplication [14] as they require a significant amount of additional data memory which is a sparse resource on the targeted light-weight devices.

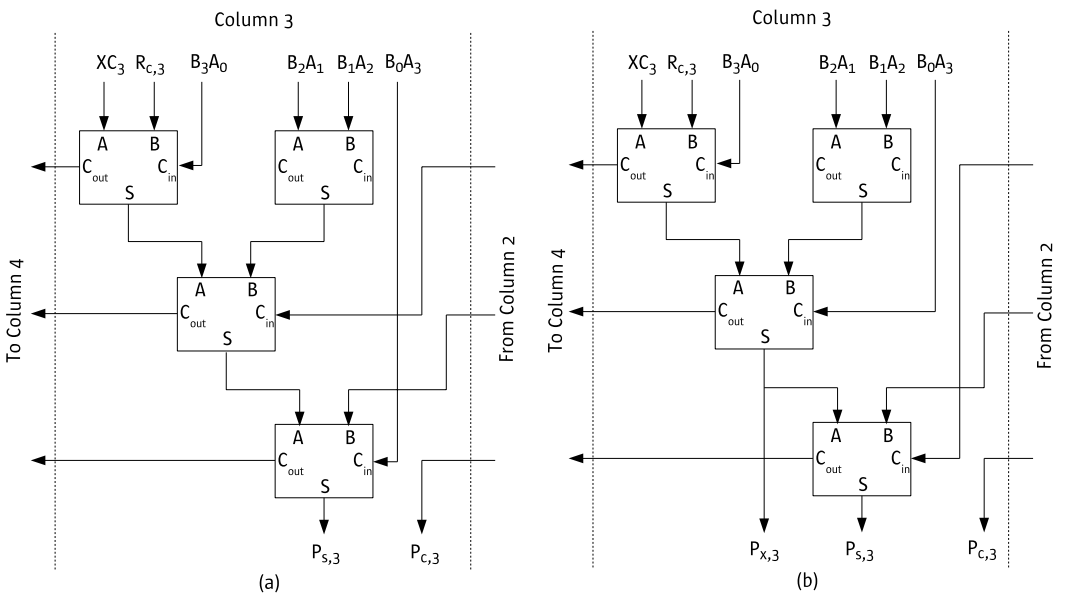

Figure 7. Conventional (a) and modified (b) CSA tree column.

For implementing ECC point multiplication over fields $GF(2^m)$, we use the point multiplication algorithm proposed by López and Dahab in [16]. This algorithm represents points on the elliptic curves with projective coordinates to avoid costly inversion operations [16] as explained in Section 3.4. We apply curve-optimized reduction to implement modular arithmetic. That is, we make use of the fact that the considered curves use sparse irreducible polynomials. With these polynomials, it is possible to implement reduction simply through a small number of shift and addition operations. Again, we chose not to resort to any optimization techniques such as window methods [11] that increase the footprint of the implementation.

Table 1 summarizes performance numbers for ECC and RSA calculations on the ATmega128 8-bit microprocessor running at 8 MHz. We split the table vertically by grouping ECC and RSA key sizes such that they represent comparable security strengths. The first group is made up of ECC secp160r1, ECC sect163r1, and RSA-1024, and the second group includes ECC secp224r1, ECC sect233r1, and RSA-2048.

The performance numbers for the ECC point multiplications over the prime integer fields $GF(p)$ and the RSA modular exponentiations were taken from [6]. For ECC curves over binary polynomial fields, we consider three implementation options: 1) an unmodified architecture with XOR multiplication implemented in software; 2) a data path equipped with a dual-field multiplier supporting a MULX instruction; and 3) a data path equipped with a dual-field multiplier supporting a MULACCX instruction.

The ATmega128 microprocessor executes multiplication instructions in two cycles. Referring to Figures 5 and 6, we can assume that the CSA tree takes one cycle and the CPA takes another cycle. Since the XOR multiplication only relies on computations by the CSA tree, MULX or MULACCX can likely be executed in a single cycle. We, therefore, give performance numbers for both 1- and 2-cycle implementations of these instructions.

Our performance measurements show that ECC point multiplication over $GF(2^m)$ executes in less than one second: We measured 0.29 s for 163-bit ECC and 0.81 s for 233-bit ECC assuming a 1-cycle MULACCX instruction. These numbers give ECC over $GF(2^m)$ a

Table 1. ECC and RSA execution times.

Algorithm	mult.	time [s]	data mem [byte]	instr mem [byte]
ECC sect163r1	XOR	4.14	239	8767
ECC sect163r1 with MULX (2 cycles)	XOR	0.46	239	2967
ECC sect163r1 with MULX (1 cycle)	XOR	0.40	239	2967
ECC sect163r1 with MULACCX (2 cycles)	XOR	0.35	239	2869
ECC sect163r1 with MULACCX (1 cycle)	XOR	0.29	239	2869
ECC secp160r1	integer	0.81	282	3682
RSA-1024 public-key $e = 2^{16} + 1$	integer	0.43	542	1073
RSA-1024 private-key with CRT	integer	10.99	930	6292
ECC sect233r1	XOR	10.98	338	7180
ECC sect233r1 with MULX (2 cycles)	XOR	1.28	338	2888
ECC sect233r1 with MULX (1 cycle)	XOR	1.12	338	2888
ECC sect233r1 with MULACCX (2 cycles)	XOR	0.97	338	2816
ECC sect233r1 with MULACCX (1 cycle)	XOR	0.81	338	2816
ECC secp224r1	integer	2.19	422	4812
RSA-2048 public-key $e = 2^{16} + 1$	integer	1.94	1332	2854
RSA-2048 private-key with CRT	integer	83.26	1853	7736

substantial performance advantage of up to a factor of 2.7 over $GF(p)$. In comparison with RSA, ECC over $GF(2^m)$ supported by a 1-cycle MULACCX instruction offers a dramatic performance gain of two orders of magnitude when comparing 233-bit ECC and 2048-bit RSA private key operations, and a factor of 2.4 considering RSA public-key operations.

Comparing memory requirements we find that the implementations for $GF(2^m)$ require less data and instruction memory than implementations for $GF(p)$ or RSA. With respect to the RSA implementations, the $GF(2^m)$ implementations require about half as much memory for instructions, and an even smaller fraction of memory for program data.

Our measurements confirm the importance of an optimized multiple-precision multiplication. For example, when executing a 163-bit ECC point multiplication, 60.14 % of the execution time is spent on multiple-precision multiplications alone [4]. To gain a better understanding of the instruction distribution, we have assembled Table 2 that quantifies the instructions executed by the multiple-precision multiplication routines (excl. reduction). We have analyzed the following three cases: 1) 163-bit modular multiplications for ECC point multiplications over $GF(2^m)$; 2) 160-bit modular multiplications for ECC point multiplications over $GF(p)$; 3) 512-bit Montgomery multiplications for RSA modular exponentiations [5]. The numbers for cases 2) and 3) are taken from [6]. With respect to case 1), we consider both a 1-cycle and a 2-cycle XOR multiplier. For each of these three cases, we examine the instruction decomposition for: a) a multiplier that provides MUL and MULX, respectively; b) a multiplier that supports MULACC and MULACCX, respectively. We notice that the majority of instructions executed in the absence of a multiply-accumulate instruction are additions [6]. These instructions are mostly eliminated if a multiply-accumulate instruction is supported. The highest gain is achieved for 163-bit multiplications in $GF(2^m)$ where support for 1-cycle MULACCX reduces execution time by 48.36 %.

[4] This measurement assumes hardware support for a 2-cycle MULX instruction.

[5] We use the hybrid multiplication method described in [6] with the following column widths d and resulting operand widths w: 1) $d = 7$, $w = 168$; 2) $d = 5$, $w = 160$; 3) $d = 6$, $w = 528$.

[6] Addition in $GF(2^m)$ translates into XOR operations.

Table 2. Instruction decomposition for multiple-precision multiplications.

ATmega128 with MUL(X) instruction									
			163x163 $GF(2^m)$			**160x160** $GF(p)$		**512x512 Montg.**	
Instruction Type	**Opcodes**	**Cyc./ Instr.**	**Instr. Cycles**	**% 1-cyc.**	**% 2-cyc.**	**Instr. Cycles**	**%**	**Instr. Cycles**	**%**
Addition, XOR	ADD/ADC/XOR	1	882	48.36	38.94	1360	43.79	29766	45.67
Multiplication	MUL(X)	1/2	441/882	24.18	38.94	800	25.76	17556	26.94
16-bit reg. move	MOVW	1	0	0.00	0.00	335	10.79	7262	11.14
Data loads	LD/LDI	2	252	13.82	11.13	334	10.75	6169	9.47
Data stores	ST	2	84	4.61	3.71	80	2.58	524	0.80
Jumps	RJMP/IJMP	2	2	0.11	0.09	66	2.12	0	0.00
Function calls/rets	CALL/RET	4	72	3.95	3.18	0	0.00	1452	2.23
Other			91	4.99	4.02	131	4.22	2442	3.75
Total			1824/2265	100.00	100.00	3106	100.00	65171	100.00
Time @ 8 MHz				0.23ms	0.28ms		0.39ms		8.15ms
ATmega128 with MULACC(X) instruction									
			163x163 $GF(2^m)$			**160x160** $GF(p)$		**512x512 Montg.**	
Instruction Type	**Opcodes**	**Cyc./ Instr.**	**Instr. Cycles**	**% 1-cyc.**	**% 2-cyc.**	**Instr. Cycles**	**%**	**Instr. Cycles**	**%**
Addition	ADD/ADC	1	0	0.00	0.00	320	16.11	6292	15.87
Multiply-accumulate	MULACC(X)	1/2	441/882	46.82	63.77	960	48.34	20328	51.27
16-bit reg. move	MOVW	1	0	0.00	0.00	15	0.76	2	0.01
Data loads	LD/LDI	2	252	26.75	18.22	334	16.82	6169	15.56
Data stores	ST	2	84	8.92	6.07	80	4.03	524	1.32
Jumps	RJMP/IJMP	2	2	0.21	0.14	66	3.32	0	0.00
Function calls/rets	CALL/RET	4	72	7.64	5.21	0	0.00	1452	3.66
Other			91	9.66	6.58	211	10.62	4884	12.32
Total			942/1383	100.00	100.00	1986	100.00	39651	100.00
Time @ 8 MHz				0.12ms	0.17ms		0.25ms		4.96ms
Time reduction				48.36%	38.94%		36.06%		39.16%

8 Related Work

There are only a few publications that examine low-cost implementations of public-key cryptosystems on light-weight devices that do not rely on a dedicated cryptographic coprocessor.

In [19], Woodbury, Bailey and Paar describe an ECC implementation for optimal extension fields (OEFs) $GF((2^8 - 17)^{17})$ that are known for enabling efficient reduction [8]. They measured 1.83 s on an 8-bit microprocessor 8051 running at 12 MHz for executing an ECC point multiplication using a fixed base point - this is the operation needed to generate signatures.

An instruction set extension for multiple-precision arithmetic on RISC-based smart cards has been described in [10]. This extension only tackles integer arithmetic, while we present a multiplier circuit and corresponding instruction set extensions that support both fields $GF(p)$ and $GF(2^m)$.

Gura *et al.* are describing an ECC implementation over fields $GF(p)$ for an 8-bit microprocessor in [6]. They measured an execution time of 0.81 s for a 160-bit ECC point multiplication on the Atmel ATmega128 microprocessor running at 8 MHz.

Pietiläinen evaluated the relative performance of RSA and ECC on smart cards in [18].

9 Conclusions

We have shown that public-key cryptography becomes a viable option even on the smallest 8-bit microprocessors. More specifically, we have described simple extensions to a data path and instruction set of a general-purpose microprocessor that enable ECC point multiplication operations over $GF(2^m)$ in under one second. These extensions require considerably fewer chip resources than traditional approaches based on on-chip cryptographic coprocessors.

It is worth pointing out, that our implementation relies on standardized elliptic curves. Given the performance results, we see no need to avert the standards already in place by inventing new techniques or using non-standard fields such as optimal extension fields [8].

Our performance analysis clearly shows the performance advantage made possible by ECC point multiplications over fields $GF(2^m)$. We have found that the performance ratio of ECC point multiplications over $GF(2^m)$ to ECC point multiplications over $GF(p)$ grows with decreasing processor word size and increasing key size. The reason is that multiple-precision squaring scales linearly for $GF(2^m)$ as the word size of the processor is decreased, whereas it scales quadratically for $GF(p)$.

ECC point multiplications over fields $GF(2^m)$ not only offer a performance advantage, the nature of the chosen algorithm [16] makes our implementation robust against attacks using power or timing analysis. With this algorithm, the execution time is constant and, thus, no information about the cryptographic keys is revealed.

We also find that the relative performance advantage of ECC over RSA increases as the word size of the processor decreases. Though both the RSA modular exponentiation and ECC point multiplication are dominated by multiple-precision modular multiplications whose complexity grows quadratically with decreasing processor word size, the ECC point multiplication contains a larger fraction of operations such as addition and optimized reduction that have linear complexity. And as mentioned before, in the case of ECC point

multiplications over $GF(2^m)$ squarings also contribute to the operations with linear complexity.

We measured a dramatic speedup of 2.4 and 102.8 for 233-bit ECC over 2048-bit RSA public-key key and private-key operations, respectively. These numbers prove that strong security, even exceeding today's security levels, can be provided even on light-weight devices.

References

[1] Chang Shantz, S.: *From Euclid's GCD to Montgomery Multiplication to the Great Divide.* Sun Microsystems Laboratories Technical Report TR-2001-95, `http://research.sun.com/`, June 2001.

[2] U.S. Department of Commerce, National Institute of Standards and Technology, *Digital Signature Standard (DSS)*, Federal Information Processing Standards Publication FIPS PUB 186-2, January 2000.

[3] Certicom Research, *SEC 2: Recommended Elliptic Curve Domain Parameters*, Standards for Efficient Cryptography, Version 1.0, September 2000.

[4] H. Eberle, N. Gura, S. Chang Shantz, *Generic Implementations of Elliptic Curve Cryptography using Partial Reduction*, Proceedings 9th ACM Conference on Computers and Communications Security, November 18-22, 2002, Washington, DC, pp. 108-116.

[5] H. Eberle, N. Gura, S. Chang Shantz, *A Cryptographic Processor for Arbitrary Elliptic Curves over $GF(2^m)$*, Proceedings IEEE 14th Int. Conference on Application-specific Systems, Architectures and Processors, June 24-26, 2003, The Hague, The Netherlands, pp. 444-454.

[6] N. Gura, A. Patel, A. Wander, H. Eberle, S. Chang Shantz, *Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs*, to be presented at the Workshop on Cryptographic Hardware and Embedded Systems CHES 2004, August 11-13, 2004, Boston, MA.

[7] L.-S. Au, N. Burgess, *Unified Radix-4 Multiplier for $GF(p)$ and $GF(2^n)$*, Proceedings IEEE 14th Int. Conference on Application-specific Systems, Architectures and Processors, June 24-26, 2003, The Hague, The Netherlands, pp. 226-236.

[8] D. Bailey, C. Paar, *Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms*, Advances in Cryptography — CRYPTO '98, Lecture Notes in Computer Science, vol. 1462, Springer-Verlag, 1998, pp. 472-485.

[9] L. Dadda, *Some Schemes for Parallel Multipliers*, Alta Frequenza, vol. 34, 1965, pp. 349-356.

[10] J. Großschädl, *Instruction Set Extension for Long Integer Modulo Arithmetic on RISC-Based Smart Cards*, Proceedings 14th Symposium on Computer Architecture and High Performance Computing, October 28 - 30, 2002, Vitoria, Brazil, pp. 13-19.

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

[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, *System Architecture Directions for Networked Sensors*, 9th Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), November 12-15, 2000, Cambridge, MA, pp. 93-104

[13] A. Satoh, K. Takano, *A Scalable Dual-Field Elliptic Curve Cryptographic Processor*, IEEE Transactions on Computers, vol. 52, no. 4, April 2003, pp. 449-460.

[14] Karatsuba, A., Ofman, Y.: Multiplication of Many-Digital Numbers by Automatic Computers. Doklady Akad. Nauk, SSSR 145, 293-294. Translation in Physics-Doklady 7, 595-596, 1963.

[15] J. Kahn, R. Katz, K. Pister, *Next Century Challenges: Mobile Networking for "Smart Dust"*, 5th Annual ACM/IEEE Int. Conference on Mobile Computing and Networking (MOBICOM), August 15-19, 1999, Seattle, WA, pp. 271-278.

[16] J. López, R. Dahab, *Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation*, 1st Int. Workshop on Cryptographic Hardware and Embedded Systems, CHES 1999, Lecture Notes in Computer Science 1717, Springer-Verlag 1999, Worcester, Massachusetts, August 12-13, 1999 pp. 316-327.

[17] P. Montgomery, *Modular Multiplication without Trial Division. Mathematics of Computation*, vol. 44, no. 170, April 1985, pp. 519-521.

[18] H. Pietiläinen, *Elliptic curve cryptography on smart cards*, Helsinki University of Technology, Faculty of Information Technology, Master's Thesis, October, 2000.

[19] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd Edition, John Wiley and Sons, 1996.

[20] C. Wallace, *A Suggestion for a Fast Multiplier*, IEEE Transaction on Electronic Computers, vol. 13, 1960, pp. 14-17.

[19] A. Woodbury, D. Bailey, C. Paar, *Elliptic Curve Cryptography on Smart Cards without Coprocessors*, 4th Smart Card Research and Advanced Applications (CARDIS2000) Conference, Bristol, UK, September 2000.