Hardware/Software Co-Design of Elliptic-Curve Cryptography for Resource-Constrained Applications

Andrea Höller, Norbert Druml,
Christian Kreiner and Christian Steger
Institute of Technical Informatics
Graz University of Technology
{andrea.hoeller, norbert.druml,
christian.kreiner, steger}@tugraz.at

Tomaz Felicijan
Infineon Technologies Austria
Design Center Graz
tomaz.felicijan@infineon.com

ABSTRACT
ECC is an asymmetric encryption providing a comparably high cryptographic strength in relation to the key sizes employed. This makes ECC attractive for resource-constrained systems. While pure hardware solutions usually offer a good performance and a low power consumption, they are inflexible and typically lead to a high area.
Here, we show a flexible design approach using a 163-bit GF(2m) elliptic curve and an 8-bit processor. We propose improvements to state-of-the-art software algorithms and present innovative hardware/software codesign variants. The proposed implementation offers highly competitive performance in terms of performance and area.

Keywords
Elliptic Curve Cryptography, RFID, Embedded Devices

1. INTRODUCTION
Radio Frequency Identification (RFID) is a popular technology when it comes to automatically identifying people and goods wirelessly. In contrast to simple ID transmission applications, security relevant applications require cryptographic-based authentication. Public-key cryptography provides a simpler key management than symmetric cryptography, since no secret key is required on the readers side [20]. Thus, public-key cryptography is more reasonable in open-loop applications. Compared to conventional public-key algorithms like RSA, ECC can achieve the same level of security with shorter key sizes. However, since the resources of an RFID tag are extremely limited, the implementation of ECC on such tags is a challenging task.
To achieve a reasonable computation time, previous implementations of ECC on RFID were mainly based on pure hardware solutions. However, development teams need flexible systems in order to react quickly to changing demands of the market. Flexibility can be achieved by using a lightweight microprocessor.
We present several options of partitioning hardware and software to offer a good runtime performance. The main contributions of this paper are:

- It proposes an algorithm for binary field multiplication in software that achieves a good performance with low storage requirements.
- It introduces a novel method of hardware/software co-design of ECC by presenting a small hardware extension that significantly speeds up the software implementation.
- It offers approaches for further hardware extensions like instruction set extensions and a coprocessor for binary multiplication.

2. THE RFID-TAG ARCHITECTURE
The target application of the presented ECC architectures are RFID tags which can range from a low-capability device (e.g. for pet identification) to a powerful contactless smartcard (e.g. for biometric passports). The modules of a typical tag IC are an analog front end, a digital control unit and a Non-Volatile Memory (typically realized as EEPROM). The area of RFID tag ICs range between 0.25 and 10 mm^2 [12].

2.1 The Microprocessor
In order to implement the control unit an 8-bit proprietary processor offering a reduced instruction set for RFID is used. Compared to hardwired state machines, the programmable framework supports a more efficient development of RFID products. Since the processor is very small ($\sim$2.5 kGE) and energy-efficient (the average power consumption is between 11.6μW/MHz and 26μW/MHz), the processor is ideal for resource-constrained applications.
The processor is based on the Harvard architecture with separate pathways for instructions and data and can be classified as a load-and-store architecture. The processor features 16 general purpose registers (GPRs) and 30 instructions. All basic instructions of a microprocessor such as binary operations, addition, subroutine functions, and branch-conditions are supported. Although there is a shift-left instruction, there is no shift-right functionality. Furthermore, the processor does not feature a multiplication operation. All instructions, can be executed within one clock cycle, except memory accesses to ROM, which require two cycles.
Primarily, the processor was designed for RFID communication protocols according to ISO/IEC 15603 or ISO/IEC 14443. However, it can also be used for more advanced calculations as shown in this paper.

3. ECC DESIGN DECISIONS
Designing an ECC-based system involves decisions on the following hierarchical levels: Security protocol, elliptic curve arithmetic and field arithmetic including field operations. Our design decisions on each of these levels are outlined in the next two sections. Since the field operation multiplication accounts for the majority of runtime, the remainder of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
DAC '14 June 01 - 05 2014, San Francisco, CA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ACM 978-1-4503-2730-5/14/06...$15.00.
http://dx.doi.org/10.1145/2593069.2593148

this paper focuses on proposals for effective multiplication implementation methods.

3.1 Security Protocol and Elliptic Curve Arithmetic

Our implementation establishes a one-way authentication of RFID tags and is based on the work of Bock et al. [5]. The authentication is implemented by a challenge-response protocol requiring one point multiplication on the tag. A Montgomery multiplication operating on projective coordinates including several protections against side channel attacks establishes the point multiplication as presented in [5].

3.2 Field Arithmetic

Standards define elliptic curves over prime fields *GF(p)* or binary fields $GF(2^m)$ [6]. The hardware support of processors offering a multiplier, favours the usage of prime fields [10, 22, 6]. However, it has been demonstrated that software-based ECC can achieve better performance using binary fields [19]. Since our processor does not feature a hardware multiplier and the usage of binary fields eases further hardware accelerations [6], we use binary fields $GF(2^m)$ in polynomial base representation.
The parameter size m is 163, thus one element is stored in 21 words. Throughout this document $A[i]$ refers to the i^{th} word of an array representing the binary vector representation of an polynomial $a(z)$. Whereby $A[0]$ stores the lower and $A[20]$ stores the higher coefficients.
Note, the protocol and point multiplication do not require an inversion operation. Thus, to compute the Montgomery multiplication the following field operations are required: addition, reduction, squaring and multiplication.

3.2.1 Addition

This operation only requires a word-wise XOR (binary addition) of both addends.

3.2.2 Reduction

Modulo reduction $c(z)=c_d(z) \bmod f(z)$ reduces the output of a field multiplication $c_d(z)$ with a field size of $(2m-1)$ to a size of m using a irreducible polynomial $f(z)=z^m+r(z)$.
Basically, the reduction algorithm goes through all coefficients c_j of $c_d(z)$ that have to be reduced and adds $z^j r(z)$ to $c_d(z)$, if the coefficient is one. The elliptic curve parameters we use define an irreducible polynomial where two 8-bit words are needed to store $r(z)$.
For acceleration, we use two lookup tables (LUTs) to store precalculated additions of shifted $r(z)$. They require 288 bytes in total. Since shifts $z^j r(z)$, where $j \geq 8$, can be achieved with array indexing, the reduction can be calculated without shifts during runtime.

3.2.3 Squaring

Squaring can be achieved by inserting zeros between two consecutive bits, as described in [6]. To square the element A, every nibble of the lower words ($A[0]$ to $A[10]$) is expanded to a 21-word element with a 16-byte LUT.
The expansion of the remaining words of A would result in words, which have to be reduced afterwards. However, we take advantage of the fact that when squaring an element, every second bit is zero and implement an interleaved reduction similar to the standard reduction. We use two 32-byte LUTs to reduce the number of shifts and additions during runtime.

3.2.4 Multiplication

The runtime of the binary field multiplication represents the main factor for the overall performance. Thus, below we will describe novel methods for accelerating the binary field multiplication.

Input: $a(z)$ and $b(z)$ of degree at most $m-1$
Output: $c(z) = a(z) \cdot b(z)$ of degree at most $2m-2$
Compute all $B_u = u(z) \cdot b(z)$ where $deg\{u(z)\} < w$
for $k \leftarrow (8/w)$ **downto** 0 **do**
  for $j \leftarrow 0$ **to** 20 **do**
    $u = (u_{w-1}, ..., u_1, u_0)$, where u_i is bit $(wk+i)$ of $A[j]$
    for $i \leftarrow 0$ **to** 20 $C[i+j] \leftarrow C[i+j] \oplus B_u[i]$
  end for
  if $(k \neq 0)$ $C \leftarrow C \cdot z^w$
end for

Figure 1: Left-to-right binary field multiplication. Adapted from [6].

3.3 Enhanced Binary Field Multiplication

A well-known approach for the binary field multiplication is the *left-to-right (l-t-r) comb* method as shown in Fig. 1. The algorithm calculates $C = A \cdot B$ by processing w bits of every word of A at a time and requires precalculation of multiples of B. The choice of w comes with a trade-off between memory requirements and performance. In general, the number of precalculated elements equals $2^w - 1$. For example, if $w=2$ the products $B_1 = B$, $B_2 = 2_d \cdot B$, $B_3 = 3_d \cdot B$ are precalcuated and stored. Note that $B_0 = 0 \cdot B$ does not need to be stored, since it is always zero. To accelerate the calculation, the window size could be increased to $w=4$. However, this would require the storage of 15 elements requiring 315 byte RAM, which is often not appropriate for a resource-constrained device such as an RFID tag.
To achieve a good performance with low storage requirement, we propose a novel enhancement of the *l-t-r comb* method. The idea is to perform fewer precalculations and calculate more during runtime by ignoring the last term of $u(z)$ in the precalculation phase. Only those $B_u = u'(z) \cdot b(z)$, satisfying $deg\{u(z)\} < w$ and $u'(z) = \{u_{w-1}z^{w-1} + ... + u_2 z^2 + u_1 z\}$ are determined and stored. Put simply, only the B_u, where u is even, are considered for precalculation. Additionally, $B_1 = B$ is stored. This reduces the number of stored elements to 2^{w-1}. For example, choosing a window size of $w=4$, requires only eight elements (168 bytes) to be stored in RAM.
The enhanced algorithm is shown in Fig. 2. The precalculation procedure can be designed to reduce the number of required shift operations by combining already calculated elements. If $w=4$ the outer loop has to be executed two times. If the processed nibble u of A is even, it is only necessary to add the corresponding B_u to the accumulator C. If u is odd, u' is determined by setting the last bit of u to zero. Thereafter, B_u is read from RAM and additionally B_1 is added to C. In simple terms, if for example $u = 7_d$, then $B_7 = 7_d \cdot B$ is calculated as follows: $B_7 = 6_d \cdot B + B$. The term $B_6 = 6 \cdot B$ is determined by reading B_6 from RAM. Thus, only the additional addition of B_1 is calculated during runtime.

3.4 Binary Field Multiplication using Virtual Addressing

Here we explore an innovative small-footprint hardware extension approach to speed up the multiplication by reducing the number of pointer calculations and memory accesses. Typically, dedicated coprocessors or instruction set extensions accelerate ECC. We propose using the idea of virtual addressing to design a hardware accelerator.
Many procedures needed for the calculation of ECC access memory consecutively and hence require many pointer calculations. Loop unfolding can reduce the number of pointer calculations, but involve a large increase of code size.
Virtual addressing allows for the use of static coded addresses (virtual addresses), without increasing the code size significantly. This can be achieved by inserting a virtual address logic between the processor and the external RAM as shown in Fig. 3. Whenever the microprocessor accesses

Input: $a(z)$ and $b(z)$ of degree at most $m-1$
Output: $c(z) = a(z) \cdot b(z)$ of degree at most $2m-2$
$B_1 \leftarrow B$; $B_2 \leftarrow B \cdot z$; $B_4 \leftarrow B_2 \cdot z$; $B_8 \leftarrow B_4 \cdot z$; $B_6 \leftarrow B_4 \oplus B_2$;
$B_{10} \leftarrow B_8 \oplus B_2$; $B_{12} \leftarrow B_8 \oplus B_4$; $B_{14} \leftarrow B_{12} \oplus B_2$
for $k \leftarrow 1$ **downto** 0 **do**
  for $j \leftarrow 0$ **to** 20 **do**
    if $(k = 1)$ $u = (A[j] \gg 4))$ **else** $u = A[j]$
    if bit 0 of u is set and $u \neq 1$ **then**
      for $i \leftarrow 0$ **to** 20 **do**
        $u' = u'$&0x0E
        $C[i+j] \leftarrow C[i+j] \oplus B'_u[i] \oplus B_1[i]$
      end for
    else if $u \neq 0$ **then**
      for $i \leftarrow 0$ **to** 20 $C[i+j] \leftarrow C[i+j] \oplus B_u[i]$
    end if
  end for
  if$(k \neq 0)$ $C \leftarrow C \cdot z^4$
end for

Figure 2: *Enhanced left-to-right comb* multiplication with windowsize $w = 4$.

Figure 3: Principle of virtual addressing.

an address in a virtual address range, the virtual address logic translates the address into a physical address. The microprocessor can influence the address mapping by setting configuration parameters. This is achieved by writing the desired value of the parameter to a predefined address. Registers within the virtual address logic store the parameters.

The basic idea of virtual addressing can be illustrated with the acceleration of a binary addition A=A+B. A straightforward implementation would keep the addresses of *A[0]* and *B[0]* in registers. After loading, XORing and storing the words, the address registers would be increased to process the next words. These pointer additions can be outsourced to hardware by using two 21-byte virtual elements VE_A and VE_B. Parameters called par_A and par_B could indicate to which physical addresses the virtual elements should point. For example, if par_A is one, then VE_A points to the first 21-byte of the RAM, if par_A is two, then VE_A points to the second 21-byte, and so on.

In the following, we show how virtual addressing can be used to speed up the *l-t-r comb* multiplication. We illustrate the approach by using a word size of *W=8*, a window size of *w=4* and 13 available GPRs. However, the method could also be adapted for other conditions.

3.4.1 Virtual Address Logic

The virtual addressing concept includes one 22-byte virtual element. Five parameters are used for the address mapping. The determination of u and calculation of the start address of B_u (see Fig. 1) can be outsourced to the virtual addressing logic by using two parameters: *element* and *addr_mode*. The parameter *element* is set to the processed word of A and *addr_mode* defines which bits are used to determine u as shown in Equation 1. Thus, u indicates which window is currently being processed.

$$u = \begin{cases} element[7:4] & \text{when } addr\_mode = 0 \\ element[3:0] & \text{when } addr\_mode = 1 \end{cases} \quad (1)$$

To achieve a shifting of B_u, the parameters *neg_offset* and *offset* are used according to Equation 2.

$$VE[i] \xrightarrow[\text{maps to}]{} B_u[i + \text{offset} - \text{neg\_offset}], i = \{0, 1, ..., 21\} \quad (2)$$

Furthermore, there is a parameter *shiftC* to shift the accumulator C as shown in Equation 3.

$$C[i] \xrightarrow[\text{maps to}]{} C[i + \text{shiftC}], i = \{0, 1, ..., 40\} \quad (3)$$

3.4.2 Binary Field Multiplication Algorithm

Fig. 5 shows the proposed algorithm for implementing a *l-t-r comb* multiplication using the virtual addressing features. First, the required multiples of B are precalculated and stored in RAM. To keep the address logic simple, we assume that all B_u elements are subsequently stored in RAM. The virtual addressing mechanism does not influence the precalculation step.

The processing of the first window starts by setting the parameter *addr_mode* to zero. Then, the algorithm has to go through all words of A and add the corresponding B_u to C. The determination of u depending on the currently processed word of A is outsourced to the virtual addressing logic by setting the parameter *element* to the currently processed word of A. This causes a mapping of the virtual element to the currently required B_u.

These additions represent the most expensive part of the multiplication, due to the high number of required word-wise additions (see Fig. 6). Nearly all words of C, which are affected during one addition, are manipulated again by the successive addition. Using virtual addressing it is possible to perform the operations on these bytes of C, which are altered most frequently, with registers: Instead of loading values from memory and storing the altered content back to the same position, all operations which target these addresses are performed with predefined registers.

The algorithm using the virtual addressing contains two subroutines implementing successive word-wise additions with the virtual element. It is possible to jump to every single word-wise addition. For example, to process $A[0]$, all word-wise additions of the first subroutine are executed. The virtual addressing logic is configured so that $VE[0]$ points to $B_u[0]$, $VE[1]$ points to $B_u[1]$ and so on.

When processing $A[1]$, $C[0]$ is not affected (see Fig. 6). Thus the algorithm jumps into the second word-wise addition of the first subroutine. Then, 20 word-wise additions are performed starting with the addition of $VE[1]$ to $C[1]$. It is then necessary to add $B_u[0]$ to $C[1]$. Thus, the parameter *neg_offset* is set to one before calling the subroutine. This causes the virtual address logic to map $VE[1]$ to $B_u[0]$, $VE[2]$ to $B_u[1]$ and so on. The last word-wise addition is realized with the second subroutine by calling *ADD_B2_1* and setting the parameter *offset* to 9. $VE[12]$ then points to B_u*[12-neg_offset+offset]*=B_u*[20]*. Thus, $B_u[20]$ is added to $C[21]$.

The remaining words of A are processed in a similar way. The pattern of processing the first and second half of A is very similar to that shown in Fig. 6. The processing of the words *A[10]* to *A[20]* can use the same code of the implementation realizing the processing of *A[0]* to *A[9]*, when shifting the accumulator C. This is realized by setting the parameter *shiftC* before performing the remaining additions. After processing the first window, the parameter *addr_mode* is set to one to process the second window of every word of A, and the whole procedure is repeated.

3.4.3 Advantages of Virtual Addressing

As stated above, the word-wise additions represent the most time consuming task of the binary field multiplication, since they are executed very often. As shown in Fig. 6, 882 such word-wise additions are required for processing both windows. A straight forward assembler implementation of a word-wise addition would require six instructions as shown in Fig. 4. Outsourcing the pointer calculations to hardware reduces the number of required instructions to four.

Additionally, the approach also reduces the number of memory accesses. By storing some intermediate results in 13 GPRs, it is possible to eliminate the need for memory accesses for 538 word-wise additions. This means that only two instruction are required.

Consequently, this saves about 2,100 instructions per multiplication while introducing a small overhead of about 100

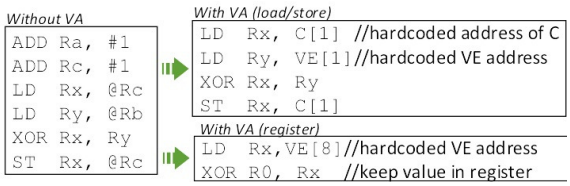

Figure 4: Assembler implementation of a word-wise addition with/without virtual addressing.

Precalcuate all B_u
$A_{ptr} \leftarrow$ address of $A[0]$
ADDR_MODE $\leftarrow 0$
call PROCESS_WINDOW
ADDR_MODE $\leftarrow 1$
call PROCESS_WINDOW
return

PROCESS_WINDOW:
Reset all registers
SHIFTC $\leftarrow 0$
for $k \leftarrow 0$ **to** 9 **do call** MULT_LOOP
Store and load registers from/to C ▷ see Fig. 6
SHIFTC $\leftarrow 10$
for $k \leftarrow 0$ **to** 10 **do call** MULT_LOOP
return

MULT_LOOP:
ELEMENT $\leftarrow$ value stored in A_{ptr}
$A_{ptr} \leftarrow A_{ptr} + 1$
NEG_OFFSET $\leftarrow$ k
call ADD_B1_[k]
if $k \neq 0$ **then**
  OFFSET $\leftarrow 9$
  call ADD_B2_[k]
return

ADD_B1_0: $C[0] \leftarrow C[0] \oplus VE[0]$ ▷ Subroutine 1
ADD_B1_1: $C[1] \leftarrow C[1] \oplus VE[1]$
...
ADD_B1_7: $C[7] \leftarrow C[7] \oplus VE[7]$
ADD_B1_8: $R0 \leftarrow R0 \oplus VE[8]$
ADD_B1_9: $R1 \leftarrow R1 \oplus VE[9]$
...
ADD_B1_20: $R12 \leftarrow R12 \oplus VE[20]$
return

ADD_B2_10: $C[30] \leftarrow C[30] \oplus VE[21]$ ▷ Subroutine 2
ADD_B2_9: $C[29] \leftarrow C[29] \oplus VE[20]$
...
ADD_B2_1: $C[21] \leftarrow C[21] \oplus VE[12]$
OFFSET $\leftarrow 0$
return

Figure 5: Algorithm implementing the *l-t-r comb* method using virtual addressing.

instructions for setting the parameters for virtual addressing (see Fig. 5). The principles of virtual addressing could also be adapted to speed up other cryptographic algorithms.

3.5 Instruction Set Extension

A well-known approach for accelerating a software implementation is to expand the microprocessor to support specific instructions. We examined two additional instructions, which lead to a significant performance improvement: a shift-right instruction and an instruction to load a value from RAM and XOR it with a register. Both operations are executed during one clock cycle. These extensions accelerate ECC and other cryptographic algorithms such as AES as well.

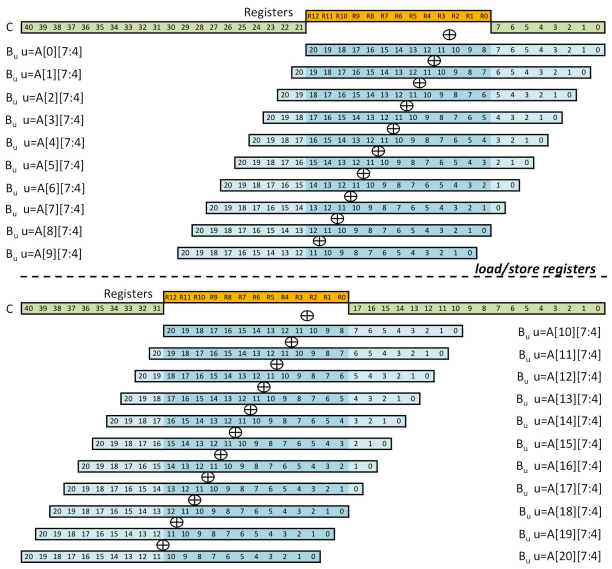

Figure 6: Illustration of additions stated in Fig. 5 for processing the first window. To process the second window, the bits [3:0] of the currently processed A are used to determine u.

3.6 Lightweight Coprocessor for Binary Field Multiplication

Next, we present a low-cost coprocessor for outsourcing the whole field multiplication and reduction to hardware.

3.6.1 Coprocessor Design

To keep the communication overhead low, the coprocessor communicates with the RAM directly via Direct Memory Access. The coprocessor first reads two factors from RAM, then performs a multiplication, and finally stores the result back to the RAM. The addresses of the two factors and the result are provided by the microprocessor. During the calculation of the coprocessor, the microprocessor pauses.
Typical ECC coprocessors in literature have relatively high area requirements, since they offer partial multiplications with high bit lengths (i.e. *mxm*-bit [18] or *mx1*-bit [14]). In general, a *mxn*-bit binary multiplier requires $m \cdot n$ AND and *(m-1)·(n-1)* XOR gates.
We propose using a $4x8$-bit multiplier and calculating an $8x8$-bit multiplication in two cycles with seven additional XOR gatters (see Fig. 7). The multiplication algorithm used executes about 440 $8x8$ bit multiplications. Hence, one multiplication takes about 440 cycles longer using a $4x8$-bit multiplier. However, compared to a conventional $8x8$-bit multiplier this saves the area of 32 AND and 21 XOR gatters.

3.6.2 Binary Field Multiplication and Reduction

The coprocessor features an own control logic to calculate the binary field multiplication and reduction. The availability of additional hardware influences the choice of the multiplication algorithm. If partial hardware multiplication is supported, it is proposed to use Comba's method [9].
The difference to the *l-t-r* multiplication is the order in which the partial products are generated. Comba's method determines each word of the result C at a time proposed in literature and includes two nested loops: the first one calculates the words $C[20]$ to $C[40]$ and the second one determines the words $C[0]$ to $C[19]$. Only one store operation is required for every word of the result. This order of calculation favours an interleaved reduction. This means that the higher words, which would require a reduction, are not

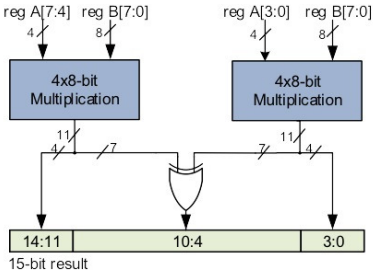

Figure 7: Construction of an 8x8-bit multiplication with a 4x8-bit multiplication.

stored in the accumulator. They are directly reduced, which is implemented in hardware requiring only 13 XOR-gatters. For more information about Comba's method, we refer to [9] and [6].

4. RESULTS

The previously presented methods have been implemented and simulated. The software implementations were coded in assembler and the hardware accelerators were implemented at Register Transfer Level using the SystemVerilog language. For area comparison the variants were synthesized in standard 220nm CMOS technology. The result of the synthesis represented the space needed for the standard cell area. To take place and route into account assumed 20% were added to the cell area.

Table 1 summarizes the performance and area requirements of the previously presented methods. Note that the given areas do not include the microprocessor.

Table 1 illustrates that the binary field multiplication is the most time-consuming field operation and is thus the determining factor for the overall performance. It accounts for about 80% of the execution time without hardware accelerators. The low-area hardware extensions lead to significant performance improvements of the field multiplication.

Compared to the standard *l-t-r comb* method with a window size of *w=4*, the *enhanced l-t-r comb* method with a window size of *w=4* needs 148 bytes less RAM. However, it requires 105 bytes more RAM than the standard *l-t-r comb* method with a window size of *w=2*, while decreasing the execution time by 25%.

The next implemented variant is a combination of the virtual addressing (VA) concept described above and the *enhanced l-t-r comb* algorithm. Therefore, only slight adaptations of the algorithm shown in Figure 5 were required. Adding the virtual address logic causes a small additional area overhead (est. +1kGE), but the performance further improves by 26%.

The two additional instructions lead to a negligible area overhead and improve the computation performance even further by 27% to 5.1 MCycles.

The coprocessor offers the best performance/area trade-off. Since the availability of additional hardware features changed the choice of the field multiplication algorithm, precomputation is no longer required. This approach reduced the size of the required RAM to almost a half. Furthermore, the coprocessor lowered the ROM storage requirement by a factor of 3.5. This is due to the outsourcing of the multiplication logic to dedicated hardware. Hence, software code for the field multiplication is no longer necessary. We determined the area of the coprocessor with synthesis, which reported 1.41 kGE combinational and 0.64 kGE non-combinational area. We assumed 20% additional area for routing overhead, which leads to a 2.13 kGE area-footprint. Additionally, VA logic for field addition was used (est. 0.3kGE). In sum, the area reduction due to lower storage requirements was larger than the additional introduced area of the coprocessor. As a consequence, this solution requires the smallest area-footprint. In addition, due to the usage of dedicated hardware to calculate the field multiplication, the performance of the coprocessor variant is far above that of previous partitioning methods. Compared to the fastest pure software approach a speed up of 3.5 was reached.

The results show that hardware extensions can almost always accelerate the execution time at the expense of area. An exception is the coprocessor variant. Due to the algorithmic change, this solution offers both the fastest runtime and the smallest area.

5. RELATED WORK

In recent years, many authors showed that RFID is ready for hardware-based ECC. The most notable implementations were presented by Batina et al. [3], Hein et al. [11], Kumar et al. [15], Lee et al. [16], Wolkersdorfer et al. [23], and Bock et al. [5]. They mainly use binary fields, require between 10.4 and 23.8 kGE area, and consume between 32.4 and 500 $\mu W/MHz$ of power.

A great deal of research has also been done in ECC on 8-bit microprocessors. Most publications target Wireless Sensor Networks (WSNs) and use the ATmega128 [2] processor. Malan et al. investigated the feasibility of ECC over binary fields in WSNs [17]. However, their implementation was quite slow, requiring about 2,510 MCycles per authentication. Guara et al. showed in [10] that ECC offers significant performance advantages compared to RSA on 8-bit architectures. Yan and Shi proposed a sophisticated inversion algorithm to speed up the ECC calculation [24]. Seo et al. [19] proposed an approach for reducing the number of memory accesses, which was then implemented assembly-optimized in [13]. In [22], Wenger et al. built a low-area processor (6.5 kGE) for RFID applications and presented AES, Grøstl, and ECC implementations.

Several papers describe how to accelerate software-based ECC with dedicated hardware such as a coprocessor (e.g., [1, 14, 4]) or instruction set extensions (e.g., [20, 7, 9, 8]).

The presented ECC hardware/software architecture compares favourably with works described in the literature. Table 2 highlights different 8-bit ECC implementations. The implementations over prime fields exploit the ATmega128's hardware multiplier. Guara et al. [10] reached considerable performance results. However, they used a non-adjacent form method for point multiplication. Wenger et al. [22] presented a clone of the ATmega128 targeting resource- constrained RFID tags. The silicon footprint of their proposed processor is 6.5 kGE, which is almost two times larger than the processor we used. Our approach requires comparable memory resources, but achieves much better performance results.

The ECC calculations presented in [21] and [13] were also performed over binary fields and could not take advantage of the hardware multiplier. The implementation of Kargl et al. [13] is comparable to ours, since they use a similar field and point multiplication method. By fully utilizing the 32 registers available on the ATmega128, they reduced the number of memory accesses and reached a runtime of 6.1 MCycles. Thus, the execution time is faster than our pure software approach, but requires significantly more ROM. Furthermore, we had only half as many GPRs at our disposal.

By using our presented hardware extensions, we achieve a notable runtime performance, small memory requirements, and maintain a high level of flexibility.

6. CONCLUSION

ECC is well suited to security related resource-limited applications. However, current smart card and RFID tag solutions often focus on pure inflexible hardware ECC designs. We offer design proposals for implementing ECC using a lightweight 8-bit RFID processor. An effective enhancement of a state-of-the art software-based binary field multiplication algorithm is presented. Furthermore, a novel ECC

Table 1: Area and performance comparison of implementation variants

Implementation Variant	*Code Size (ROM)*		*RAM*		*Ext.*	***Area***	*Binary Field Operations [Cycles]*				***Mont. Mult.***
	[Byte]	*[kGE]*	*[Byte]*	*[kGE]*	*[kGE]*	***[kGE]***	*Add.*	*Squar.*	*Red.*	*Mult.*	***[MCycles]***
L-t-r mult. w=2	3,205	3.41	318	4.1	-	**7.51**	150	1,560	580	9,750	**12.1**
Enh. l-t-r mult. w=4	3,123	3.32	423	5.46	-	**8.92**	150	1,560	580	7,640	**9.4**
Enh. l-t-r mult. w=4 & VA	3,840	4.09	423	5.46	1	**10.55**	90	1,540	580	5,280	**7.0**
Enh. l-t-r mult. w=4 VA & ISE	3,594	3.83	423	5.46	1	**10.29**	75	1,070	500	3,820	**5.1**
Coprocessor & ISE	1,023	1.09	214	2.76	2.43	**6.18**	75	1,070	-	1,830	**2.8**

Table 2: Comparison to 8-bit ECC implementations available in literature

Implementation Variant	*GF*	*ROM*	*RAM*	*Runtime*
		[kByte]	*[Byte]*	*[MCycles]*
Guara et al.[10]	p_{160}	3.6	280	6.48
Wenger et al.[22]				
Slowest version	p_{160}	3.86	384	35.1
Fastest version	p_{160}	7.76	384	13.0
Szczechowiak et al.[21]	2^{163}	32.4	1741	16.0
Kargl et al [13]	2^{167}	11	>588	6.1
Our implementation				
Pure software	2^{163}	3.05	423	9.7
VA and ISE	2^{163}	3.51	423	5.1
Coprocessor	2^{163}	1.02	214	2.8

hardware/software partitioning approach using virtual addressing is introduced. Additional hardware/software partitioning variants are outlined and evaluated.
Our approach is highly competitive regarding performance and area. The fastest variant requires about 0.2s@13.56MHz to calculate a challenge. This is about 4.6 times faster than the most similar solution using a microprocessor to calculate ECC on an RFID tag [22]. We showed that a software-based development of ECC is practical for applications, such as brand protection, which are not very time critical.
Our future work includes an analysis of the proposed implementation methods in the light of side-channel attacks and power consumption.

Acknowledgment

We would like to thank our industrial partner Infineon Technologies Austria AG, in particular all members of Infineon's PDC team for their great support and the Austrian Federal Ministry for Transport, Innovation, and Technology, which funded the project META[:SEC:] under the FIT-IT contract 829586.

7. REFERENCES

[1] H. Aigner, H. Bock, M. Hütter, and J. Wolkerstorfer. A low-cost ECC coprocessor for smartcards. *CHES*, 2004.
[2] Atmel Corp. *8-bit Microcontroller with 128K Bytes In-System Programmable Flash: ATmega 128*, 2004.
[3] Batina et al. Hardware architectures for public key cryptography. *Integration, the VLSI journal*, 2003.
[4] G. Bertoni, L. Breveglieri, and M. Venturi. Power aware design of an elliptic curve coprocessor for 8 bit platforms. In *PerCom*. IEEE, 2006.
[5] H. Bock, M. Braun, M. Dichtl, E. Hess, J. Heyszl, W. Kargl, H. Koroschetz, B. Meyer, and H. Seuschek. A Milestone Towards RFID Products Offering Asymmetric Authentication Based on Elliptic Curve Cryptography. *Invited talk at RFIDsec*, 2008.
[6] S. V. D.R. Hankerson and A. Menezes. *Guide to Elliptic Curve Cryptography*. 2004.
[7] W. Drescher, K. Bachmann, and G. Fettweis. VLSI architecture for datapath integration of arithmetic over GF(2m) on digital signal processors. In *Acoustics, Speech, and Signal Processing*, 1997.
[8] H. Eberle, A. Wander, N. Gura, S. Chang-Shantz, and V. Gupta. Architectural extensions for elliptic curve cryptography over $GF(2^m)$ on 8-bit microprocessors. *ASAP*, 2005.
[9] Großschädl et al. When Instruction Set Extensions Change Algorithm Design: A Study in Elliptic Curve Cryptography. 2005.
[10] Gura et al. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. *CHES*, 2004.
[11] D. Hein, J. Wolkerstorfer, and N. Felber. ECC is Ready for RFID–A Proof in Silicon. In *Selected Areas in Cryptography*, 2009.
[12] U. Kaiser, C. Paar, J. Pelzl, D. Rappe, W. Schindler, A. Weimarskirch, and T. Wollinger. Auswahlkriterien fuer kryptographische Algorithmen bei Low-Cost-RFID-Systemen, 2005.
[13] A. Kargl, S. Pyka, and H. Seuschek. Fast arithmetic on ATmega128 for elliptic curve cryptography. *context of the SMEPP project*, 2008.
[14] Koschuch et al. Hardware/software co-design of elliptic curve cryptography on an 8051 microcontroller. *CHES*, 2006.
[15] S. Kumar and C. Paar. Are standards compliant elliptic curve cryptosystems feasible on RFID? In *Workshop on RFID Security*, 2006.
[16] Lee et al. Elliptic-curve-based security processor for RFID. *IEEE Transactions on Computers*, 2008.
[17] D. Malan, M. Welsh, and M. Smith. A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography. In *Sensor and Ad Hoc Communications and Networks*, 2004.
[18] S. Okada, N. Torii, K. Itoh, and M. Takenaka. Implementation of elliptic curve cryptographic coprocessor over $GF(2^m)$ on an FPGA. In *CHES*, 2000.
[19] S. Seo, H. Dong-Guk, H. Kim, and H. Seokhie. TinyECCK: Efficient Elliptic Curve Cryptography Implementation over $GF(2^m)$ on 8-Bit MICAz Mote. *IEICE transactions on information and systems*, 2008.
[20] Z. Shi and H. Yan. Software implementations of elliptic curve cryptography. *International Journal of Network Security*, 2008.
[21] Szczechowiak et al. NanoECC: Testing the limits of elliptic curve cryptography in sensor networks. *WSNs*, 2008.
[22] E. Wenger, T. Baier, and J. Feichtner. JAAVR: Introducing the Next Generation of Security-enabled RFID Tags. *Euromicro Conference on DSD*, 2012.
[23] J. Wolkerstorfer. Scaling ECC Hardware to a Minimum, 2005. Slides of a talk given at Workshop CRASH 2005, Leuven.
[24] H. Yan and Z. Shi. Studying software implementations of elliptic curve cryptography. In *Information Technology: New Generations*, 2006.