A Cryptographic Processor for Low-Resource Devices: Canning ECDSA and AES Like Sardines

Michael Hutter[1], Martin Feldhofer[1], and Johannes Wolkerstorfer[2]

[1] Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria
{Michael.Hutter,Martin.Feldhofer}@iaik.tugraz.at
[2] xFace, Plüddemanngasse 39, 8010 Graz, Austria
Johannes.Wolkerstorfer@xface.at

Abstract. The Elliptic Curve Digital Signature Algorithm (ECDSA) and the Advanced Encryption Standard (AES) are two of the most popular cryptographic algorithms used worldwide. In this paper, we present a hardware implementation of a low-resource cryptographic processor that provides both digital signature generation using ECDSA and encryption/decryption services using AES. The implementation of ECDSA is based on the recommended $\mathbb{F}_{p192}$ NIST elliptic curve and AES uses 128-bit keys. In order to meet the low-area requirements, we based our design on a sophisticated hardware architecture where a 16-bit datapath gets heavily reused by all algorithms and the memory is implemented as a dedicated RAM macro. The proposed processor has a total chip area of 21 502 GEs where AES needs only 2 387 GEs and SHA-1 requires 889 GEs.

Keywords: Cryptographic Processor, ECDSA, ECC, AES, SHA-1, ASIC Implementation, Low-Resource Constraints.

1 Introduction

In a world where an innumerable amount of pervasive devices communicate with each other, the need for security increases heavily. Cryptographic services like secure symmetric and asymmetric authentication as well as confidentiality build the basis for contactless security applications like access control, mobile payment, and product authentication.

Most of the published hardware implementations of cryptographic services optimize a single algorithm or even a part of it and often do not account for higher-level protocols and applications. Turning such cryptographic primitives into a working product turns out to require a multiple of resources in the end. In this paper, we investigate the implementation of a cryptographic processor for low-resource devices. We present a complete integrated solution which is based only on standardized algorithms and protocols.

In particular, using standardized algorithms and protocols with an appropriate level of security is important to assure the interoperability between devices

C.A. Ardagna and J. Zhou (Eds.): WISTP 2011, LNCS 6633, pp. 144–159, 2011.

and to allow reuse of existing infrastructures in back-end applications. Even in the very cost-sensitive market, people get more and more convinced that standardized solutions are inevitable. Two of the most important standardized algorithms are the Elliptic Curve Digital Signature Algorithm (ECDSA) [23] and the Advanced Encryption Standard (AES) [21]. ECDSA, which is for example used for secure identification in e-passports, generates digital signatures for message and entity authentication. AES is the successor of the Data Encryption Standard (DES) and today the most frequently used symmetric block cipher for encryption and authentication.

Hence, we target the implementation of our cryptographic processor on these two algorithms. The reason why we have chosen to implement both algorithms in one module is that the public-key scheme ECDSA can be used for offline authentication in open-loop applications while AES is much faster when the verifier has online access especially in closed-loop scenarios. Furthermore, with our approach of reusing components like the memory and the controlling engine we want to demonstrate that these high-security algorithms can be migrated to very resource-constrained devices such as mobile devices, embedded systems, wireless sensors, and RFID devices.

In this paper, we present the first ASIC hardware implementation of a cryptographic processor that is able to perform both the ECDSA using the NIST elliptic curve over $\mathbb{F}_{p192}$ and the AES (encryption and decryption) with 128-bit keys. Our implementation targets low-resource devices which implies fierce requirements concerning chip area (costs) and power consumption (due to a possible contactless operation). We meet the ambitious design goals by using a sophisticated hardware architecture where the main components memory, datapath, and controlling engine are reused by all implemented algorithms. Using a 16-bit datapath with a multiply-accumulate unit and a dedicated RAM macro instead of a flip-flop based memory minimizes the chip area. Next to several algorithmic-level improvements, we present a very efficient arithmetic-level implementation of a modular multiplication with interleaved NIST reduction over $\mathbb{F}_{p192}$. The entire processor needs 21 502 GEs where 2 387 GEs are required to support AES and 889 GEs are needed for SHA-1. This is because AES and SHA-1 reuse several components of our processor such as the microcontroller and the common memory.

The article is structured as follows. Section 2 summarizes related work on AES and elliptic-curve hardware implementations. An overview of the system is given in Section 3 and the hardware architecture is described in detail. Arithmetic-level implementation are given in Section 4 where we describe the NIST modular multiplication. Section 5 shows the implemented algorithms such as SHA-1, AES, and ECDSA. Results of our work are presented in Section 6. Conclusions are drawn in Section 7.

2 Related Work

There exist many articles that describe hardware architectures for AES and elliptic-curve based algorithms. One landmark paper that reports a low-resource

implementation of AES is due to M. Feldhofer et al. [4] in 2004. Their implementation needs 3 595 GEs and performs a 128-bit encryption within 1 016 clock cycles. P. Hämäläinen et al. [7] presented an encryption-only AES architecture in 2006. Their design needs only 3 100 GEs. Similar results have been reported also by J.-Kaps et al. [14] and M. Kim [15] who presented an encryption-only AES implementation with around 4 000 GEs.

In view of elliptic-curve cryptography (ECC) there exist several implementations that propose efficient hardware architectures for scalar multiplication, e.g. S. Kumar et al. [17] and L. Batina et al. [1]. Architectures with implementations of also higher-level protocols have been proposed by A. Satoh et al. [24] and J. Wolkerstorfer [27] who proposed a dual-field ECC processor for low-resource devices. Y. K. Lee et al. [18] and D. Hein et al. [9] presented an ECC co-processor over binary fields $\mathbb{F}_{2^{163}}$. The work of Lee integrates a tiny microcontroller for higher-level arithmetics while the work of Hein includes a digital RFID front-end supporting the ISO 18000-3-1 standard protocol. ECDSA implementations have been realized by J. Wolkerstorfer [27], F. Fürbass et al. [5], and E. Wenger et al. [26]. They based their design on prime-field arithmetics to support ECDSA.

Our work is based on an ECDSA implementation of M. Hutter et al. [10]. We extended the work by implementing AES-128 (supporting encryption and decryption) as a main contribution and show that it can be integrated into the processor with low resources. We further give a detailed description of the arithmetic-level and algorithmic-level implementation of the processor and discuss the results in Section 6.

3 System Overview

The implementation of our proposed processor is based on the recommended NIST Weierstrass elliptic curve over $\mathbb{F}_{p192}$. This has mainly two reasons. First, our processor should be as flexible and scalable as possible while keeping the required chip area low. A processor over $\mathbb{F}_p$ allows us to support different protocols and algorithms on the processor without the need of any additional logic circuits. However, the costs for that choice are a lower performance compared to binary-extension field processors. Second, fixing the implementation to a standardized elliptic curve provides interoperability with existing applications like X.509 public-key infrastructures (PKI), citizen cards, and e-passports. Furthermore, it allows several optimizations in hardware like the NIST modular reduction [8] to gain additional performance.

We decided to implement a 16-bit architecture. During our investigations, it has shown that a 16-bit data width provides an optimum for reducing the chip area and the power consumption while keeping the required number of clock cycles within limitations.

3.1 Hardware Architecture

In order to design a low-resource processor, we minimized the required hardware resources by reusing components like the memory and the controller for all

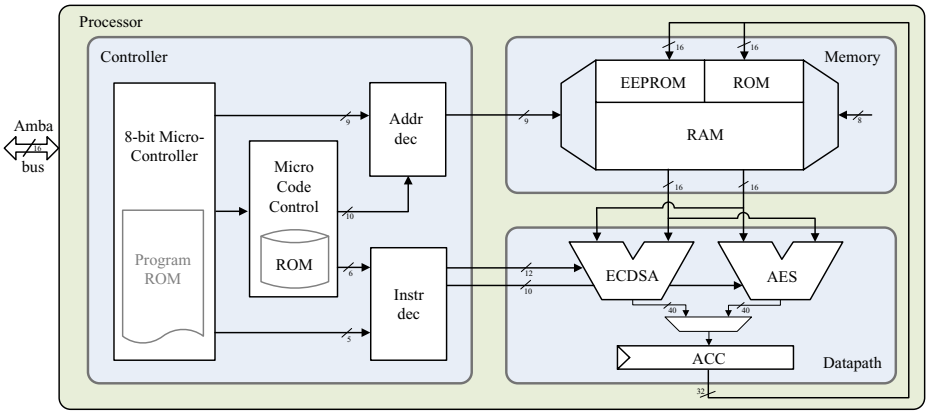

Fig. 1. Architecture of the Cryptographic Processor

implemented algorithms. Especially for the AES this means that the overhead is very low because the ECDSA dominates the memory requirements and the controlling effort.

The cryptographic processor consists of three main components as depicted in Figure 1. The first component is the controller, which is responsible for sequencing the desired algorithms and the generation of the control signals for the memory and the datapath unit. The second module is the memory, which holds data during computation, constants like curve parameters, and also non-volatile data like the private key (for ECDSA) and the secret key (for AES). The third module is the datapath, which performs the arithmetic and logic operations for ECDSA and AES.

The memory of the processor can be accessed by a memory-mapped I/O. Via an AMBA interface it is possible to write and read data to and from the RAM (e.g. the message to sign or the generated signature) but also to access the EEPROM or the instruction register. In very complex algorithms and protocols like ECDSA with implicit SHA-1 calculation and random-number generation, the controlling effort in terms of design complexity and chip area gets more and more dominant. Hence, we investigated a totally new concept where a micro-controlled approach makes the implementation more flexible but keeps also the hardware complexity low compared to dedicated finite-state machines.

Memory Unit. The memory unit comprises the three types RAM, ROM, and EEPROM in a 16-bit linear addressable dual-port memory space. The 128×16-bit dual-port RAM is realized as a dedicated macro block. This halves the chip area of this memory resource. In detail, ECDSA needs 7×192 bits for calculating the point multiplication, one 192-bit value to store the message that has to be signed, and one 192-bit value for the ephemeral key k. Additionally, we reserved 192 bits for storing the seed that is used in both ECDSA and AES to generate the needed random numbers. The ROM circuit stores 128 16-bit

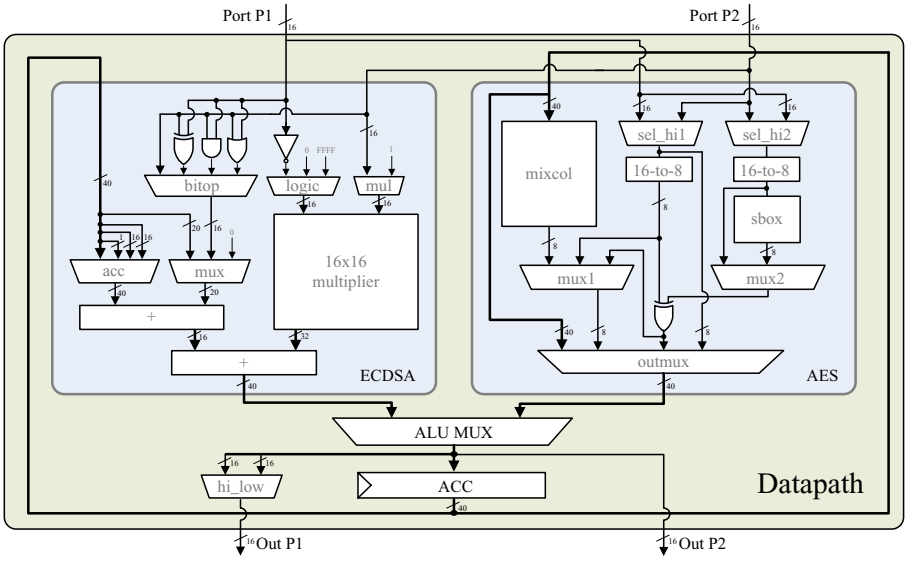

Fig. 2. Architecture of the ECDSA and AES Datapath

constants like ECC parameters, SHA-1, and AES constants. It is implemented as an unstructured mass of standard cells. The EEPROM stores non-volatile data like the ECDSA private key, the public-key certificate, the AES secret key, and potentially other user-specific data up to 4K bits, which can be written in a personalization phase or during the protocol execution.

Datapath Unit. The datapath of our processor is shown in Figure 2. It is mainly composed of an ECDSA and AES datapath. Both datapaths share one single 40-bit accumulator register which pursues the strategy of reusing components for ECDSA and AES. The 40-bit register is used as accumulator for ECDSA (multiply-accumulate unit) as well as intermediate storage for AES.

The AES datapath is mainly composed of an S-box submodule, a MixColumns submodule, five multiplexers, one XOR gate, and two 16-to-8 bit converters. The converters are used since we decided to implement the AES operations with 8 bits instead of 16. The remaining 8 bits are used to store random values which are required to perform dummy operations[1]. The AES datapath has been implemented similar to the work of M. Feldhofer et al. [4]. For the S-box operation, we calculated the substitution values using combinatorial logic instead of a look-up table, which reduces the number of additional gates. Furthermore, the MixColumns operation has been realized as an individual submodule which generates one byte of the AES *State* using one single clock cycle. The ShiftRows and AddRoundKey operations, in contrast, have been realized without expensive logic circuits. These operations

[1] Dummy operations are types of *hiding* countermeasure techniques against side-channel attacks (see [19] for more details).

need several controlling signals and one XOR gate. The AES constant *Rcon* has been externally stored in the ROM memory. In addition, we integrated an *operand-isolation* technique (also often referred to as *sleep logic*) to reduce the power consumption of the processor. If AES in enabled, the operands for AES get isolated from the ECDSA datapath. This eliminates unnecessary power dissipation and reduces the power consumption of the processor by about 13 %.

The ECDSA datapath is mainly composed of a 16 × 16-bit multiplier, two 40-bit adders, and five multiplexers. For low-area reasons, we decided to use a 16-bit multiply-accumulate (MAC) architecture to perform a finite-field multiple-precision multiplication. For this, partial products are calculated and accumulated in the common register to perform a multiplication. The implemented algorithm for the multiple-precision modular multiplication is described in Section 4.

SHA-1 is an integral part of ECDSA and is used to hash digital messages. Thus, we decided to integrate all needed components to perform SHA-1 operations into the ECDSA datapath. These are four additional 16-bit logic gates, i.e. AND, OR, XOR, and NOT. The logic operations are directly connected to port A and port B of the entire datapath. The *bitop* and *mux* multiplexer are then used to output the result of the appropriate operation. For a detailed description of the SHA-1 standard see the FIPS-180-3 [22] standard.

Low-Resource Microcontroller. A sophisticated two-layer approach was necessary to efficiently implement the controlling of the cryptographic processor. The generation of control signals for various irregular algorithms and protocols, which require in total several 100 000 clock cycles, is very complex. The highest layer of the controller comprises an 8-bit microcontroller with a highly optimized instruction set. It performs higher-level functions like protocol handling, point multiplication, and invocation of round functions for SHA-1 and AES, for instance, due to its ability for looping and subroutine calls. The advantage of having a microcontroller is that it is very flexible because of extending the functionality by simple Assembler programming. Basically the microcontroller sets up and calls certain instructions which lie in a subsequent microcode ROM table (the second control layer). For the execution of such instructions, which can take up to 102 clock cycles, the start address in the microcode ROM has to be provided. While the microcode ROM provides instructions for the datapath and the memory via the instruction and the address decoder, the microcontroller can set up the next instruction. This avoids idle cycles of the datapath during execution of the algorithm.

Our proposed microcontroller is based on a Harvard architecture, *i.e.* program memory and data memory are separated. Such a design has the advantage that the program memory can have a different word size than the data memory. The microcontroller is a Reduced Instruction Set Computer (RISC) supporting 32 instructions that have a width of 16 bits. The instructions are mainly divided into four groups: logical operations like XOR and OR, arithmetic operations like addition (ADD) and subtraction (SUB), control-flow operations like GOTO and CALL, and microcode instructions (MICRO).

4 Arithmetic-Level Implementation

Modular multiplication is the most resource-consuming operation in an ECC implementation. In fact, more than 80 % of the execution time is due to finite-field multiplications. In the following, we describe the implemented modular multiplication with interleaved NIST reduction. Modular addition and subtraction have a minor impact on the overall performance and they have been implemented according to [8].

NIST P-192 Modular Multiplication. The given algorithm is based on a product scanning (Comba) method and performs a modular multiplication using t^2 single-precision multiplications, where t represents the number of words of the processor, *i.e.* 12 in our case. In general, a multiplication by two 192-bit integers $a, b \in [0, p < 2^{Wt})$ will result in a 384-bit result c, where W represents the number of bits of a word (e.g. 16) and p represents the NIST prime $p = 2^{192} - 2^{64} - 1$. Instead of storing the 384-bit result in memory, we reduced the result during the multiplication (interleaved reduction and multiplication). Thus, no additional memory is needed for the multiplication. We make use of the following congruency [25], *i.e.*

Algorithm 1. Modular multiplication with interleaved NIST P-192 reduction.

Require: $a, b \in [0, p-1], S \in [0, 2^{3W} - 1], \varepsilon \in [0, t + t/3 - 1]$.
Ensure: $c = a * b \pmod{2^{192} - 2^{64} - 1}$.

1. S $\leftarrow$ 0.
2. **for** i **from** 0 **to** $t-1$ **do**
3. **for** j **from** $t-1$ **to** i **do**
4. S $\leftarrow$ S + A[j]*B[$i+t-j$].
5. **end for**
6. C[i] $\leftarrow$ (S mod 2^W); S $\leftarrow$ (S $\gg W$).
7. **end for**
8. C[$t-1$] $\leftarrow$ (S mod 2^W). S $\leftarrow$ 0.
9. **for** i **from** 0 **to** $t/3$ **do**
10. S $\leftarrow$ (S + C[i] + C[$i+2t/3$]).
11. C[i] $\leftarrow$ (S mod 2^W); S $\leftarrow$ (S $\gg W$).
12. **end for**
13. **for** i **from** 0 **to** $t/3$ **do**
14. S $\leftarrow$ (S + C[i] + C[$i+t/3$]).
15. C[$i+t/3$] $\leftarrow$ (S mod 2^W); S $\leftarrow$ (S $\gg W$).
16. **end for**
17. **for** i **from** 0 **to** $t/3$ **do**
18. S $\leftarrow$ (C[$i+t/3$] - C[i] - S) + C[$i+2t/3$].
19. C[$i+2t/3$] $\leftarrow$ (S mod 2^W);
20. S $\leftarrow$ (S $\gg W$) (mod 2).
21. **end for**
22. $\varepsilon \leftarrow$ S
23. **for** i **from** 0 **to** $t-1$ **do**
24. **for** j **from** 0 **to** i **do**
25. S $\leftarrow$ S + A[$i-j$]*B[j].
26. **end for**
27. **if** ($\varepsilon = t/3$) **then**
28. S $\leftarrow$ S + C[i] + ε.
29. **else**
30. S $\leftarrow$ S + C[i].
31. **end if**
32. C[i] $\leftarrow$ (S mod 2^W).
33. S $\leftarrow$ (S $\gg W$).
34. **end for**
35. $\varepsilon \leftarrow$ S
36. **for** i **from** 0 **to** $t-1$ **do**
37. **if** ($\varepsilon = t/3$) **then**
38. S $\leftarrow$ S + C[i] + ε.
39. **else**
40. S $\leftarrow$ S + C[i].
41. **end if**
42. C[i] $\leftarrow$ (S mod 2^W).
43. S $\leftarrow$ (S $\gg W$).
44. **end for**

return (c).

$$
\begin{aligned}
c \equiv \quad & c_5 2^{128} + c_5 2^{64} + c_5 \\
& + c_4 2^{128} + c_4 2^{64} \\
& + \qquad\quad c_3 2^{64} + c_3 \\
& + c_2 2^{128} + c_1 2^{64} + c_0 \qquad (\text{mod } p),
\end{aligned} \tag{1}
$$

where c_i are 64-bit integers. Equation (1) shows that c can be reduced by simple additions. The first three lines reduce the higher part $c_{high} = c_5 2^{320} + c_4 2^{256} + c_3 2^{192}$. The result is then added to the lower part $c_{low} = c_2 2^{128} + c_1 2^{64} + c_0$.

The algorithm for the modular multiplication with interleaved NIST reduction is given in Algorithm 1. First, the higher part of the 384-bit result is calculated (line 1-8). Second, the higher part is reduced by subsequent additions to the lower part of c (line 9-21). After that, the lower part of the 384-bit result is calculated and added to the already reduced result (line 23-34). In line 27-31, the carry ε is reduced by adding ε to the accumulator variable S at word index 0 (line 22) and $t/3 = 4$ (line 28). Finally, a last reduction is performed in line 35-44 to reduce the final carry ε.

The modular multiplication has been implemented as a fully unrolled microcode instruction. It needs 204 clock cycles to perform a modulo multiplication of two 192-bit numbers. Modular addition and subtraction need 31 clock cycles.

5 Algorithm-Level Implementation

5.1 The SHA-1 Algorithm

For our ECDSA processor, we decided to sign messages with a fixed length of 16 bytes. This constraint allows us to reduce the SHA-1 implementation to only one 512-bit message block W. In addition, the message padding can be implemented *a priori* by storing the length of the message in ROM. Thus, the 16-byte message can be simply copied into the RAM before signature generation. Message padding is done during the computation of ECDSA by copying the length of the message at the end of the input block W.

We implemented 13 different microcode instructions for SHA-1 and made several modifications to improve the performance (see Algorithm 2). First, since line 13 and line 20 are the same, *i.e.* $\mathrm{F} \leftarrow (\mathrm{B} \oplus \mathrm{C} \oplus \mathrm{D})$, we implemented only one microcode instruction that is invoked two times during the computation. Second, instead of copying the values of the state variables (A,B,C,D,E,T) as shown in line 25+26, we simple rotated the addressing of the variables. Thus, no additional clock cycles are needed and the addresses get shifted by the microcontroller in every loop iteration. Third, all bit-shift operations are realized by multiplication. A left shift by one (line 6) is a simple multiplication with the constant 2, a shift by five (line 8) is a multiplication by 32, and a shift by 30 (line 24) is realized by a multiplication with 16 384. Thus, no dedicated shifting unit is necessary in the datapath of the processor and the multiplier of the ECDSA datapath gets simply reused by the design. Fourth, the constants K0...K4 and the initial values for H0...H4 are stored in ROM and are loaded by the microcode instructions.

Algorithm 2. The Secure Hash Algorithm (SHA-1) [22].

Require: 512-bit block W; $H0, H1, H2, H3, H4, T, F, A, B, C, D, E \in [0, 2^{32} - 1]$.
Ensure: $h = \text{SHA-1}(W)$.

1. $A = H0; B = H1; C = H2;$
2. $D = H3; E = H4.$
3. **for** i **from** 0 **to** 79 **do**
4. **if** $(i \geq 16)$ **then**
5. W[i-3] $\oplus$ W[i-8] $\oplus$
6. W[i-14] $\oplus$ W[i-16] $\ll 1$.
7. **end if**
8. T $\leftarrow$ (A $\ll 5$) + W[i].
9. **if** $(i < 20)$ **then**
10. F $\leftarrow$ (B $\wedge$ C) $\vee$ ($\overline{B}\wedge$ D).
11. T $\leftarrow$ T + $0x5A827999$.
12. **else if** $(i < 40)$ **then**
13. F $\leftarrow$ (B $\oplus$ C $\oplus$ D).
14. T $\leftarrow$ T + $0x6ED9EBA1$.
15. **else if** $(i < 60)$ **then**
16. F $\leftarrow$ (B $\wedge$ C) $\vee$ (B $\wedge$ D) $\vee$
17. (C $\wedge$ D).
18. T $\leftarrow$ T + $0x8F1BBCDC$.
19. **else**
20. F $\leftarrow$ (B $\oplus$ C $\oplus$ D).
21. T $\leftarrow$ T + $0xCA62C1D6$.
22. **end if**
23. T $\leftarrow$ E + F.
24. B $\leftarrow$ B $\ll 30$.
25. E $\leftarrow$ D; D $\leftarrow$ C; C $\leftarrow$ B;
26. B $\leftarrow$ A; A $\leftarrow$ T.
27. **end for**
28. $H0 \leftarrow H0 + A$; $H1 \leftarrow H1 + B$;
29. $H2 \leftarrow H2 + C$; $H3 \leftarrow H3 + D$;
30. $H4 \leftarrow H4 + E$;

return $(H0, H1, H2, H3, H4)$.

Fourth, the round loop i is done by the microcontroller which also performs the branching at certain loop indices. Since the microcontroller can prepare the loop-index calculation during the execution of a microcode instruction, additional clock cycles are saved. In total, our processor needs 3 639 clock cycles to hash a 512-bit message.

5.2 The AES Algorithm

For AES, we implemented 11 microcode instructions. As already stated in Section 3.1, we extended the 16×8-bit AES *State* to 16×16-bit where 8 bits are used to store the real *State* and the other 8 bits store random values ($r_0 ... r_{15}$).

Figure 3 shows the processing of the first AES *State*. The first operations are SubBytes and ShiftRows which transform the *State* column-wise, *i.e.* four byte blocks. First, each byte of the *State* is loaded and substituted by the S-box unit. In order to implicitly shift the bytes of the *State* for the ShiftRows operation, we simply addressed the appropriate bytes in the *State*. Thus, each byte of a column is addressed accordingly (address 0, 5, 10, and 15 in our example) and substituted afterwards. The result is stored in the accumulator of the datapath. After that, the bytes are loaded from the accumulator and processed by the MixColumns operation. The output is then XORed with the key within the AddRoundKey operation. Finally, the result is stored back into the AES *State*. Since we processed the bytes of a column without the ShiftRows operation (in fact the bytes have not been shifted before MixColumns), we stored the resulting bytes in the correct position within the *State*. However, to avoid overwriting of data, we simply swap the values of the real and dummy *State*. Thus, after the first round, the dummy values are stored in the lower 8 bits and the real values are stored in the higher 8 bits of the *State*.

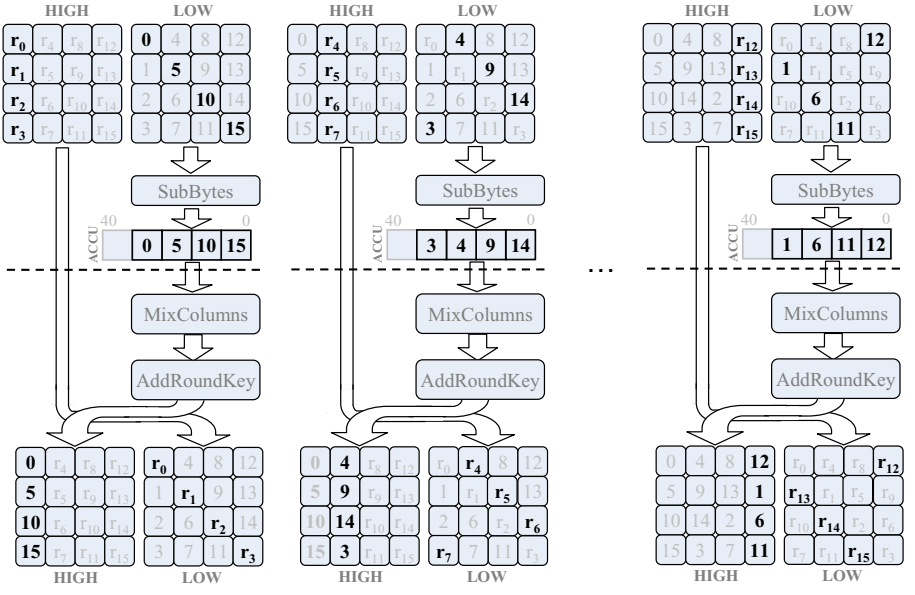

Fig. 3. The processing of the *State* in the first AES round

In order to make the implementation less attractive to side-channel attacks, we integrated several countermeasures described in the following. As a first countermeasure, we integrated dummy operations before and/or after the actual AddRoundKey operation. 16 dummy operations are performed in total where the actual operation is widespread over 17 different locations in time.

As a second countermeasure, we randomized the processing of the bytes in the AES *State* which is often referred as *byte-shuffling* countermeasure. For this, we randomized the byte position after the SubBytes operation. The transformed bytes are stored in the accumulator with a random offset. After MixColumns and AddRoundKey transformation, the offset is incorporated through the right addressing of the *State*.

As a third countermeasure, we added 16 dummy rounds to the actual rounds. In fact, we performed dummy rounds only in the first and second round and the last two rounds of AES. This has its reason in the fact that side-channel attacks need to target intermediate values which can be generated by a model with less computational effort. Targeting intermediates of higher rounds would increase the computational effort significantly to generate values for each possible key. It is therefore sufficient to consider only the first and last two rounds of AES to obtain an appropriate protection.

All implemented countermeasures are commonly used in practice and provide a state-of-the-art protection for cryptographic devices. For a more detailed description about dummy operations and shuffling (*hiding* techniques), see the work of Mangard et al. [19].

5.3 ECC Scalar Multiplication

We applied the Montgomery ladder as scalar multiplication method given in Algorithm 3. This is due to the fact that it provides security against several attacks, e.g. Simple Power Analysis (SPA) attacks [16,19] and safe-error attacks [28]. Furthermore, it allows to perform all group operations with x-coordinate only formulae [12].

We applied the idea of N. Meloni [20] and Y. Lee et al. [18] and performed the computations in a common-Z coordinate representation. The idea is to satisfy that the Z coordinate of each curve point is the same during each Montgomery ladder iteration. Only three coordinates have to be maintained during every *differential addition and doubling* operation instead of normally four, i.e. X_0, X_1, and Z. By applying the method, we did not reduce the needed number of intermediate registers but rather improved the point-multiplication performance. One Montgomery loop iteration needs therefore 12 finite-field multiplications, 4 squarings[2], 9 additions, and 7 subtractions. The memory stores three coordinates (X_0, X_1, and Z) and four intermediate values of 192 bits.

Algorithm 3. The implemented ECC scalar multiplication method based on the Montgomery ladder.

Require: Base point $P = (x_P, y_P) \in E(\mathbb{F}_{p192}), k \in [1, n-1]$, random λ
Ensure: $Q = kP$, where $Q = (x_Q, y_Q) \in E(\mathbb{F}_{p192})$
1: $(X_0, Z_0) \leftarrow (\lambda x_P, \lambda)$.
2: $(X_1, Z_1) \leftarrow Dbl(P)$.
3: $X_0 \leftarrow X_0 \cdot Z_1$, $X_1 \leftarrow X_1 \cdot Z_0$, $Z \leftarrow Z_0 \cdot Z_1$.
4: **for** $i = 190$ **downto** 0 **do**
5: $\quad (X_{k_i \otimes 1}, X_{k_i}, Z) \leftarrow DifferentialAdditionAndDoubling(X_{k_i}, X_{k_i \otimes 1}, Z)$.
6: **end for**
7: $(X_0, Y_0, Z_0) \leftarrow Yrecovery(X_0, X_1, Z, P)$.
8: **if** $Z'_0(Y'^2_0 - bZ'^2_0) \neq X'_0(X'^2_0 + aZ'^2_0)$ **abort.**
9: $x_Q \leftarrow X_0 \cdot Z_0^{-1}$.
10: **Return** (x_Q).

As a side-channel countermeasure, we applied the randomized projective coordinate (RPC) countermeasure as proposed by S. Coron [2] in 1999. Before starting a point multiplication, we generate a random number λ and performed one finite-field multiplication to randomize the affine x-coordinate of the base point x_P to obtain the randomized projective coordinates $(X_0, Z_0) = (\lambda x_P, \lambda)$.

After scalar multiplication, we perform a check if the point is still a valid point on the elliptic curve. For this, we recovered the coordinates (X_0, Y_0, Z_0) according to Izu et al. [11] and evaluated $Z'_0(Y'^2_0 - bZ'^2_0) = X'_0(X'^2_0 + aZ'^2_0)$ according to N. Ebeid and R. Lambert [3]. Finally, the projective coordinates (X_0, Z_0) are transformed back into affine coordinates by applying a finite-field inversion and multiplication, *i.e.* $x_Q \leftarrow X_0 \cdot Z_0^{-1}$.

[2] The squaring operation is realized by a simple multiplication operation.

5.4 ECDSA Implementation

After scalar multiplication, all performed operations are done modulo the prime n. The implemented ECDSA signature generation algorithm is shown in Algorithm 4. Modulo multiplication has been implemented according to the Montgomery multiplication algorithm proposed by G. Hachez and J. J. Quisquater [6]. We implemented five microcode instructions to perform the operation. The Montgomery inversion has been implemented by the algorithm proposed by B. Kaliski [13]. For that operation we implemented seven microcode instructions.

Algorithm 4. Signature Generation using ECDSA.

Require: Domain parameters $D = (q, FR, S, a, b, P, n, h)$, private key d, message m.
Ensure: Signature (r, s)
1: Select $k \in [1, n-1]$
2: Compute $Q = kP = (x_Q, y_Q)$.
3: Compute $r = x_Q \mod n$. If $r = 0$ then go to step 1.
4: Compute $e =$ SHA-1(m).
5: Compute $s = k^{-1}(e + dr) \pmod n$. If $s = 0$ then go to step 1.
6: Return (r, s).

Random numbers have been generated according to the FIPS 186-2 [23] standard. The standard describes a hash-based pseudo-random number generator that can be realized with the SHA-1 algorithm. Our decision has mainly two reasons. First, the process of random number generation is based on a standard specification and is considered to be cryptographically secure. Second, we already need a hash calculation for the message-digest calculation in ECDSA and we can simply reuse the implementation of the SHA-1 algorithm for that purpose. The prerequisite is to load a true-random seed from an external source into RAM. The random number is hashed and the message digest is stored as a seed key (XKEY). This seed key is then used in any higher-level protocol to generate any random numbers needed to provide the cryptographic service.

6 Results

We implemented our processor in a 0.35 μm CMOS technology using a semi-custom design flow with Cadence RTL Compiler as synthesis tool. The summarized results are shown in Table 1 and Table 2. The total chip area of 21 502 GEs includes datapath, ROM, RAM macro, and controller for ECDSA, SHA-1, and AES (including microcontroller and microcode control and ROM). Note that we included a standard RAM macro needing 8 727 GEs that can be further minimized using an area-optimized RAM block.

We also synthesized our processor without the AES datapath and microcode entries resulting in 19 115 GEs. This means that the integration of AES needs 2 387 GEs which is lower than existing stand-alone (and finite-state machine based) AES modules. The same has been done with SHA-1. Even though ECDSA

Table 1. Area of chip components

Component	GE
Datapath	3 393
Memory without RAM	729
RAM macro (128x16-bit)	8 727
Controller	8 653
ECDSA+SHA1+AES Total	**21 502**
Overhead of AES	**2 387**
Overhead of SHA-1	**889**

Table 2. Cycle count of operations

Component	Cycles
PRNG generation (4× SHA-1)	14 947
Point multiplication	753 393
Point-validity check	29 672
Final signing process	65 097
ECDSA Total	**863 109**
SHA-1	**3 639**
AES with shuffling	**4 529**

required SHA-1 for signing of messages, we removed any SHA-1 related implementation to validate the overhead. These are program-ROM entries, microcode instructions, decoder circuits, and the logic operations in the ECDSA datapath. After synthesis, we obtain 20 613 GEs which means that SHA-1 needs an area of 889 GEs (180 GEs for program ROM, 546 GEs for microcode instructions, and 163 GEs for the datapath).

In view of ECC, our processor needs 753 393 clock cycles for one point multiplication. The entire ECDSA signing process needs 863 109 clock cycles including side-channel and fault-attack countermeasures (RPC and point-validity check). AES with byte shuffling needs 4 529 clock cycles and 15 577 cycles with 10 dummy-round operations enabled. Note that the number of clock cycles for AES thus varies depending on the number of added dummy rounds. The more dummy rounds, the higher the security level and the higher the needed number of clock cycles. SHA-1 needs 3 639 clock cycles for hashing a 512-bit message block. The SHA-1 algorithm has also been used to generate random numbers. For ECDSA, four SHA-1 computations are performed to generate the needed random numbers which needs 14 947 clock cycles. We also evaluated the critical path of our processor and determined a maximum clock frequency of 33 MHz.

The mean current of the circuit is $485\,\mu A$ at 847 kHz and 3.3 V and has been simulated using Synopsis NanoSim. This value includes the power consumption of the entire processor including microcontroller, ECDSA, SHA-1, and AES datapath, and memory. Note that we based our design on a rather old CMOS process technology ($0.35\,\mu$m) so that further power reductions can be achieved by using a smaller process technology (for example CMOS $0.13\,\mu m$).

We compare our implementation with existing ASIC solutions over prime-field arithmetic. Since there does not exist any implementation of both ECDSA and AES within one processor, we have to compare it with ECC (or ECDSA) only implementations. The processor of F. Fürbass et al. [5] needs 23 656 GEs and 502 000 clock cycles, J. Wolkerstorfer [27] needs 23 800 GEs and 677 000 clock cycles, and M. Hutter et al. [10] need 19 115 GEs and 859 188 clock cycles. The work of A. Satoh et al. [24] needs 29 655 GEs and 4 165 000 clock cycles for the same size of prime-field arithmetic and E. Wenger et al. [26] need 11 686 GEs and 1 377 000 clock cycles. Note that a fair comparison is largely infeasible since the implementations differ in several ways, for example they do not use RAM macros and do not contain an AES implementation.

7 Conclusions

In this article we presented the first stand-alone cryptographic processor which performs ECDSA using the recommended NIST elliptic curve over $\mathbb{F}_{p192}$ and AES-128. We improved the state-of-art of building cryptographic processors for low-resource devices on the arithmetic level, on the architectural level (combined ECDSA, SHA-1, and AES module), and on the implementation level. The processor's architecture has an optimized 16-bit datapath and a controller with an integrated 8-bit microcontroller, both implemented in standard-cells. The entire chip has an area of 21 502 GEs where AES requires 2 387 GEs and SHA-1 requires 889 GEs. Currently, we are about to manufacturing the chip on a 0.35 μm CMOS process technology. The chip will be integrated in a passively powered Near Field Communication (NFC) device.

Acknowledgements

The work has been supported by the Austrian Government through the research program FIT-IT Trust in IT Systems (Project CRYPTA, Project Number 820843), and by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy).

References

1. Batina, L., Mentens, N., Sakiyama, K., Preneel, B., Verbauwhede, I.: Low-Cost Elliptic Curve Cryptography for Wireless Sensor Networks. In: Buttyán, L., Gligor, V.D., Westhoff, D. (eds.) ESAS 2006. LNCS, vol. 4357, pp. 6–17. Springer, Heidelberg (2006)
2. Coron, J.-S.: Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 292–302. Springer, Heidelberg (1999)
3. Ebeid, N., Lambert, R.: Securing the Elliptic Curve Montgomery Ladder Against Fault Attacks. In: Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography - FDTC 2009, Lausanne, Switzerland, pp. 46–50 (September 2009)
4. Feldhofer, M., Dominikus, S., Wolkerstorfer, J.: Strong Authentication for RFID Systems Using the AES Algorithm. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 357–370. Springer, Heidelberg (2004), http://springerlink.metapress.com/content/26tmfjfcju58upb2/fulltext.pdf, doi:10.1007/b99451
5. Fürbass, F., Wolkerstorfer, J.: ECC Processor with Low Die Size for RFID Applications. In: Proceedings of 2007 IEEE International Symposium on Circuits and Systems. IEEE, Los Alamitos (2007)
6. Hachez, G., Quisquater, J.-J.: Montgomery Exponentiation with no Final Subtractions: Improved Results. In: Paar, C., Koç, Ç.K. (eds.) CHES 2000. LNCS, vol. 1965, pp. 91–100. Springer, Heidelberg (2000), http://www.springerlink.com/content/n2m6w6b0kg3elaxu/

7. Hämäläinen, P., Alho, T., Hännikäinen, M., Hämäläinen, T.D.: Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core. In: Proceedings of the 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD 2006), Dubrovnik, Croatia, August 30-September 1, pp. 577–583. IEEE Computer Society, Los Alamitos (2006)
8. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography, p. 332. Springer, Heidelberg (2004)
9. Hein, D., Wolkerstorfer, J., Felber, N.: ECC Is Ready for RFID – A Proof in Silicon. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) SAC 2008. LNCS, vol. 5381, pp. 401–413. Springer, Heidelberg (2009)
10. Hutter, M., Feldhofer, M., Plos, T.: An ECDSA Processor for RFID Authentication. In: Ors Yalcin, S.B. (ed.) RFIDSec 2010. LNCS, vol. 6370, pp. 189–202. Springer, Heidelberg (2010)
11. Izu, T., Möller, B., Takagi, T.: Improved Elliptic Curve Multiplication Methods Resistant against Side Channel Attacks. In: Menezes, A., Sarkar, P. (eds.) INDOCRYPT 2002. LNCS, vol. 2551, pp. 296–313. Springer, Heidelberg (2002)
12. Joye, M., Yen, S.-M.: The Montgomery Powering Ladder. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 291–302. Springer, Heidelberg (2003), http://www.springerlink.com/content/1eupwrx4c4xayyyv/fulltext.pdf
13. Kaliski, B.: The Montgomery Inverse and its Applications. IEEE Transactions on Computers 44(8), 1064–1065 (1995)
14. Kaps, J.-P.: Cryptography for Ultra-Low Power Devices. PhD thesis, ECE Department, Worcester Polytechnic Institute, Worcester, Massachusetts, USA (May 2006)
15. Kim, M., Ryou, J., Choi, Y., Jun, S.: Low Power AES Hardware Architecture for Radio Frequency Identification. In: Yoshiura, H., Sakurai, K., Rannenberg, K., Murayama, Y., Kawamura, S.-i. (eds.) IWSEC 2006. LNCS, vol. 4266, pp. 353–363. Springer, Heidelberg (2006), http://www.springerlink.com/content/1pk7q6621xj2422r/fulltext.pdf, doi:10.1007/11908739
16. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
17. Kumar, S.S., Paar, C.: Are standards compliant Elliptic Curve Cryptosystems feasible on RFID? In: Workshop on RFID Security 2006 (RFID Sec 2006), Graz, Austria, July 12-14 (2006)
18. Lee, Y.K., Sakiyama, K., Batina, L., Verbauwhede, I.: Elliptic-Curve-Based Security Processor for RFID. IEEE Transactions on Computers 57(11), 1514–1527 (2008), doi:10.1109/TC.2008.148
19. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks – Revealing the Secrets of Smart Cards. Springer, Heidelberg (2007) ISBN 978-0-387-30857-9
20. Meloni, N.: Fast and Secure Elliptic Curve Scalar Multiplication Over Prime Fields Using Special Addition Chains. Cryptology ePrint Archive, Report 2006/216 (2006)
21. National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard (November 2001), http://www.itl.nist.gov/fipspubs/
22. National Institute of Standards and Technology (NIST). FIPS-180-3: Secure Hash Standard (October 2008), http://www.itl.nist.gov/fipspubs/
23. National Institute of Standards and Technology (NIST). FIPS-186-3: Digital Signature Standard, DSS (2009), http://www.itl.nist.gov/fipspubs/

24. Satoh, A., Takano, K.: A Scalable Dual-Field Elliptic Curve Cryptographic Processor. IEEE Transactions on Computers 52(4), 449–460 (2003), doi:10.1109/TC.2003.1190586
25. Solinas, J.A.: Generalized mersenne numbers. Technical Report CORR 99-39, Centre for Applied Cryptographic Research, University of Waterloo (1999)
26. Wenger, E., Feldhofer, M., Felber, N.: Low-Resource Hardware Design of an Elliptic Curve Processor for Contactless Devices. In: Chung, Y., Yung, M. (eds.) WISA 2010. LNCS, vol. 6513, pp. 92–106. Springer, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-17955-6_7
27. Wolkerstorfer, J.: Is Elliptic-Curve Cryptography Suitable for Small Devices? In: Workshop on RFID and Lightweight Crypto, Graz, Austria, July 13-15, pp. 78–91 (2005)
28. Yen, S.-M., Joye, M.: Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis. IEEE Transactions on Computers 49, 967–970 (2000)