# Lightweight Implementation of Hummingbird Cryptographic Algorithm on 4-Bit Microcontrollers

Xinxin Fan[1], Honggang Hu[1], Guang Gong[1], Eric M. Smith[2], and Daniel Engels[2]
[1]Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, N2L 3G1, CANADA
{x5fan, h7hu, ggong}@uwaterloo.ca

[2]AuthentiCrypt Corporation
4500 Westgrove Drive, Suite 300, Addison, TX 75001, USA
esmith@authenticrypt.com

## Abstract

*The Radio Frequency Identification (RFID) technology provides an extensible, flexible and secure measure against product counterfeiting, one of the most serious threats to the world economy. However, due to the harsh cost and power constraints of RFID tags only dedicated cryptographic engines or low-power consumption microcontrollers can be integrated into tags to implement various security mechanisms. In this contribution, we investigate the efficient software implementation of an ultra-lightweight cryptographic algorithm* Hummingbird *[5] on a zero-power 4-bit* MARC4 *microcontroller from Atmel and compare the performance of* Hummingbird *and the other ultra-lightweight block cipher* PRESENT *[4] on the same platform. Our experimental results show that after a system initialization phase* Hummingbird *can achieve about 58% faster throughput than the block cipher* PRESENT *on the 4-bit* ATAM893-D *microcontroller running at 16KHz, 500KHz and 2MHz, respectively. In particular,* Hummingbird *can process one data block with less than 12 $ms$ under a typical low power configuration of a 4-bit microcontroller such as an 1.8V supply voltage and a 500KHz clock frequency.*

## 1. Introduction

Counterfeiting is one of the most significant sources of loss in many industries including the pharmaceuticals, entertainment, consumer electronics, medical devices, retail, and IT sectors. According to the Counterfeiting Intelligence Bureau (CIB) of the International Chamber of Commerce (ICC) [9], 5 - 7% of the international trade is counterfeited and the value of the seized counterfeit products in 2007 is estimated to be over $600 billion. Although many common anti-counterfeiting measures like holograms or chemical markers have already been widely utilized by companies in their products [9], current techniques do not avail automatic verification of product authenticity and provide the required level of security. In this context, Radio Frequency IDentification (RFID) has emerged as a promising solution to combat counterfeiting since it enables automatic objects identification and complements the traditional anti-counterfeiting methods [12, 13]. While the widespread deployment of RFID systems will facilitate efficient and automated collection and management of identification information in a multitude of application domains, it also raise serious privacy and security concerns.

Considering the severely resource-constrained environment of RFID tags, many lightweight symmetric key primitives with small hardware footprint have been proposed in the literature that can be employed to implement privacy-preserving mutual authentication protocols in RFID systems. In [6, 7], Feldhofer *et al.* proposed a low-power and compact ASIC core for 128-bit-key AES with 3400 gates. Other work along the line of finding more compact AES implementations, say Hämäläinen *et al.*'s work in [8] or designing new lightweight cryptographic schemes, such as in [10] Leander *et al.* suggested a lightweight DES variant called DESL (DES Lightweight), in [4] Bogdanov *et al.* described an ultra-lightweight substitution-permutation (SP) network based block cipher PRESENT. Particularly, a serial version [11] of PRESENT can be implemented with as few as 1000 gate equivalents (GEs). More recently, Engels and Smith proposed another ultra-lightweight cryptographic algorithm called Hummingbird for RFID authenti-

cation, which is based on a hybrid mode of block cipher and stream cipher and has a hardware footprint of 1023 GEs. The description of Hummingbird encryption and authentication, together with its cryptanalysis is appeared in [5].

Besides designing compact and highly-optimized cryptographic engine for RFID tags, researchers also considered integrating low-power consumption 4-bit microcontrollers into RFID tags to assist the execution of cryptographic algorithms. In [14], Vogt *et al.* described the efficient implementation of the block cipher PRESENT [4] on the 4-bit ATAM893-D microcontroller of Atmel's MARC4 family [3] for the first time. Their experimental results showed that it is feasible to implement a lightweight cryptographic algorithm on ultra-constrained 4-bit microcontrollers. Motivated by Vogt *et al.* work [14], we consider the efficient implementation of Hummingbird [5] on 4-bit microcontrollers in this contribution. Like PRESENT, Hummingbird also uses 4 S-boxes (i.e., substitution tables with 4-bit input and 4-bit output), which makes it well suited for implementation on 4-bit microcontrollers. To provide a fair performance comparison between PRESENT and Hummingbird, we choose the same 4-bit mircrocontroller and development tools as those used in [14].

The remainder of this work is organized as follows: in Section 2 we give a brief review of the ultra-lightweight cryptographic algorithm Hummingbird, followed by the description of the used development tool for programming on the target platform ATAM893-D in Section 3. In Section 4, we describe our code size optimized implementation of Hummingbird in great detail and present our results. Finally, Section 5 concludes this contribution.

## 2. The Ultra-Lightweight Cryptographic Algorithm Hummingbird

Hummingbird, which is recently designed by Engels and Smith and reported in [5], is an ultra-lightweight cryptographic primitive for encryption and authentication in severely resource-constrained environments such as small hardware devices like passive RFID tags. It is an elegant combination of block cipher and stream cipher with 16-bit block size, 256-bit key size, and 80-bit internal state. The size of the key and the internal state of Hummingbird provides a security level which is adequate for many RFID applications. Moreover, Hummingbird has been shown to be resistant to the most common attacks to block ciphers and stream ciphers including birthday attacks, differential and linear cryptanalysis, structure attacks, algebraic attacks, and cube attacks [5]. A top-level description of Hummingbird cryptographic algorithm is shown in the following Figure 1.

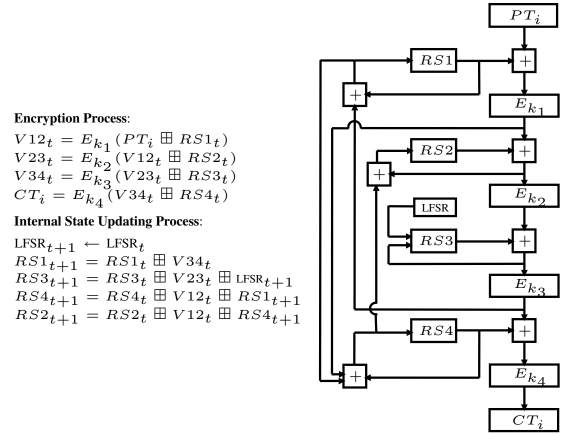


**Figure 1. A Top-Level Description of** Hummingbird **Cryptographic Algorithm**

The overall structure of the Hummingbird encryption algorithm (see Figure 1) consists of four 16-bit block ciphers $E_{k_1}, E_{k_2}, E_{k_3}$ and $E_{k_4}$, four 16-bit internal state registers $RS1, RS2, RS3$ and $RS4$, and a 16-stage Linear Feedback Shift Register (LFSR), where the 256-bit secret key $K$ is divided into four 64-bit subkeys $k_1, k_2, k_3$ and $k_4$ which are used in the four block ciphers, respectively. The encryption/decryption process of Hummingbird can be viewed as the continuous running of a rotor machine, where four small block ciphers act as four virtual rotors which perform permutations on 16-bit words. After a system initialization process with four random nonce as shown in Figure 2, a 16-bit plaintext block $PT_i$ is encrypted by first executing a modulo $2^{16}$ addition, denoted by $\boxplus$, of $PT_i$ and the content of the first internal state register $RS1$. The result of the addition is then encrypted by the first block cipher $E_{k_1}$. This procedure is repeated in a similar manner for another three times and the output of $E_{k_4}$ is the corresponding ciphertext $CT_i$. Moreover, following the internal state updating process described in Figure 1, the four internal state registers will also be updated in various and unpredictable ways.

Hummingbird employs four identical block ciphers in a consecutive manner, each of which is a typical SP-network with 16-bit block size and 64-bit key as shown in Figure 3. The block cipher consists of four regular rounds and a final round. The 64-bit subkey $k_i$ is split into four 16-bit round keys $K_1^{(i)}, K_2^{(i)}, K_3^{(i)}$ and $K_4^{(i)}$ which are used in the four regular rounds, respectively. Moreover, the final round utilizes two keys $K_5^{(i)}$ and $K_6^{(i)}$ directly derived from the four round keys (see Figure 3). While each regular round comprises of a key mixing step, a substitution layer, and a permutation layer, the final round only includes the key mixing and the S-box substitution steps. The key mixing step is implemented using a simple exclusive or operation, whereas the substitution layer is composed of

**Nonce Initialization**:
$RS1_0 = \text{NONCE}_0$
$RS2_0 = \text{NONCE}_1$
$RS3_0 = \text{NONCE}_2$
$RS4_0 = \text{NONCE}_3$

**Four Rounds Encryption**:
**for** $t = 0$ to $3$ **do**
$V12_t = E_{k_1}\left((RS1_t \boxplus RS3_t) \boxplus RS1_t\right)$
$V23_t = E_{k_2}(V12_t \boxplus RS2_t)$
$V34_t = E_{k_3}(V23_t \boxplus RS3_t)$
$TV_t = E_{k_4}(V34_t \boxplus RS4_t)$
$RS1_{t+1} = RS1_t \boxplus TV_t$
$RS2_{t+1} = RS2_t \boxplus V12_t$
$RS3_{t+1} = RS3_t \boxplus V23_t$
$RS4_{t+1} = RS4_t \boxplus V34_t$
**end for**

**LFSR Initialization**:
$\text{LFSR} = TV_3 \mid \text{0x1000}$

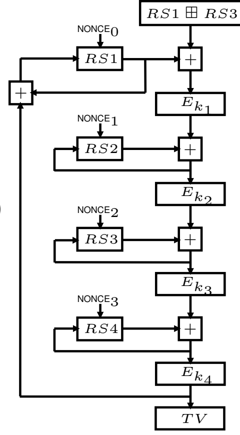


**Figure 2. The Initialization Process of** Hummingbird **Cryptographic Algorithm**

four Serpent-type S-boxes [1] with 4-bit input and 4-bit output (see Figure 3 for hexadecimal notation of four S-boxes). The permutation layer is given by the linear transform $L : \{0,1\}^{16} \rightarrow \{0,1\}^{16}$ shown in Figure 3.

Four S-Boxes Given in Hexadecimal Notation

| $x$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S_1(x)$ | 8 | 6 | 5 | F | 1 | C | A | 9 | E | B | 2 | 4 | 7 | 0 | D | 3 |
| $S_2(x)$ | 0 | 7 | E | 1 | 5 | B | 8 | 2 | 3 | A | D | 6 | F | C | 4 | 9 |
| $S_3(x)$ | 2 | E | F | 5 | C | 1 | 9 | A | B | 4 | 6 | 8 | 0 | 7 | 3 | D |
| $S_4(x)$ | 0 | 7 | 3 | 4 | C | 1 | A | F | D | E | 6 | B | 2 | 8 | 9 | 5 |

Linear Transform $L : \{0,1\}^{16} \rightarrow \{0,1\}^{16}$
$L(m) = m \oplus (m \ll 6) \oplus (m \ll 10)$,
where $m = (m_0, \cdots, m_{15})$ is a 16-bit data block.

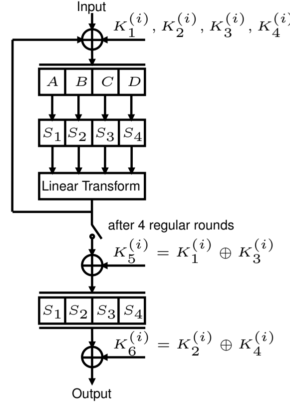


**Figure 3. The Structure of the Block Cipher in** Hummingbird **Cryptographic Algorithm**

In this paper, the compact version of Hummingbird (i.e., the substitution layer consists of four *identical* $4 \times 4$ S-Boxes $S_1$) is chosen for implementation on a 4-bit microcontroller because it further reduces memory requirements and power consumption of Hummingbird without jeopardizing its security. For more details about Hummingbird, the interested reader is referred to [5].

# 3. 4-Bit Target Platform and Development Tool

We chose the ATAM893-D [3], a member of Atmel's MARC4 family of 4-bit single-chip microcontrollers, as the target 4-bit platform due to its unique feature of high data throughput and low current consumption, which makes it a perfect candidate for energy constrained wireless applications such as keyless entry, immobilizer systems, wireless keyboards for PC and multimedia, wireless sensors as well as other applications requiring an extremely low current consumption for extended battery life. The ATAM893-D mainly consists of a MARC4 4-bit CPU core, parallel I/O ports, the Universal Timer/Counter Communication Module (UTCM) with two 8-bit programmable multi-function/timer/counters, and the clock management module with integrated RC-, 32-kHz and 4-MHz crystal oscillators. A block diagram of the ATAM893-D can be found in [3].

The high performance and low power consumption of Atmel's MARC4 microcontroller family are based on the usage of an advanced stack-based 4-bit CPU core depicted in Figure 4. The core contains 4KByte program memory (ROM), $256 \times 4$-bit data memory (RAM), 12-bit wide Program Counter (PC), four 8-bit wide RAM address registers `SP`, `RP`, `X` and `Y`, 4-bit wide Condition Code Register (CCR), 4-bit Arithmetic Logic Unit (ALU), instruction decoder and interrupt controller. The key features of the MARC4 core include [2]:

- *RISC*[1] *Architecture*: 72 simple and highly-optimized instructions are provided and most of them are single-byte instructions. Each instruction takes $0.5\mu s$ at 4MHz, which corresponds to two clock cycles per instruction.

- *HARVARD Structure*: Three independent buses (i.e., the instruction bus, the memory bus and the I/O bus) can be used in parallel to communicate between the ROM, the RAM, and attached peripherals. This feature increases the performance by allowing the processor to fetch the instruction and access the attached peripherals at the same time.

- *Power Saving Modes*: The MARC4 core provides various power-saving functions and consumes $0.1\mu$A in Deep-Sleep Mode, $0.6\mu$A in Sleep Mode, $70\mu$A in Power-down Mode and less than 1mA in Active Mode.

- *Vectored Prioritized Interrupt Levels*: The pipelined program execution, coupled with the MARC4's prioritized interrupt controller, facilitate the fast and efficient processing needed for a real-time control system.

[1] Reduced Instruction Set Computing

- *High-level Programming Language*: The MARC4 instruction set is optimized for the high-level programming language qForth. qForth has instructions for arithmetic and logical operations, control structures like loops and conditional branches, memory operations, 4-bit and 8-bit comparisons, stack operations and compiler directives.

- *Stack-based Architecture*: The MARC4 inherits a stack-based architecture where the RAM is used to construct the Expression Stack (EXP) and the Return Stack (RET), which are addressed with the Expression Stack Pointer (`SP`) and the Return Stack Pointer (`RP`), respectively. All arithmetic, I/O and memory reference operations take their operands *from*, and return their result *to* the EXP. Note that the MARC4 performs all operations with the top elements of the stack (`TOS` and `TOS - 1`) and the `TOS` register also works as an accumulator of the MARC4 core (see Figure 4). While the EXP is used for passing parameters between subroutines, the RET is used for storing return addresses of subroutines, interrupt routines and for keeping loop-index counters. Moreover, both EXP and RET can also be used as a temporary storage area. In particular, two stacks have a user-definable maximum depth.

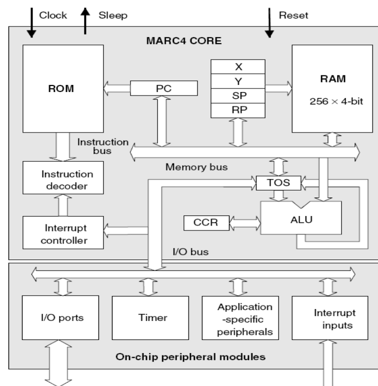


**Figure 4. The Architecture of MARC4 Core [2]**

The development tool for programming on MARC4 microcontrollers is the MARC4 Starter Kit (see Figure 5) containing the following components:

- *Hardware Components*: an *E-Lab ICP V24 Portable* programmer, an MARC4 *Programming board*, a ready-to-run application board *TEMIC T4xCx92*.

- *Software Components*: a Windows-based *editor*, an integrated qForth *compiler*, an integrated *simple* MARC4 *core simulator* (only core, no peripheral modules), an integrated *Help Function* with qForth *dictionary* and an *Atmel-wm ICP* programmer software.

**Figure 5. The Atmel MARC4 Starter Kit**

## 4. 4-Bit Implementation of the Hummingbird Cryptographic Algorithm

The MARC4 is a zero address machine with a compact and efficient instruction set. The instructions contain only the operation to be performed but no source or destination address information. Therefore, it is a formidable task to implement the Hummingbird on the target platform since all operations and function calls must be organized and scheduled according to the stack-based architecture of the MARC4 core. Figure 6 depicts the hierarchical function call graph in the implementation of Hummingbird where 11 functions have been implemented on the ATAM893-D microcontroller. The implementation details of the functions in Figure 6 will be presented in the following subsections using the qForth programming language [2].

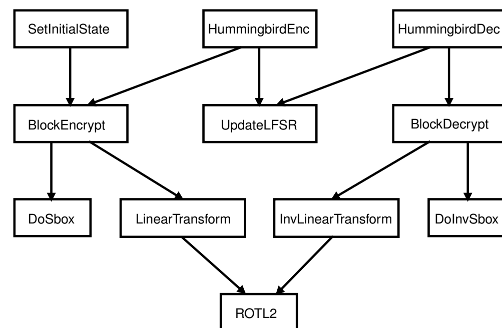


**Figure 6. The Function Call Graph in Hummingbird Implementation**

[2]To fully understand the code and its effects in this section, the reader is referred to [2].

### 4.1. Memory Allocation and Variable Initialization

Both the Expression and Return Stacks are located in RAM, whose size is variable and must be defined by the programmer. By analyzing various operations in the Hummingbird algorithm and the function call graph in Figure 6, we define the maximum depth of the EXP and the RET to be 10 and 4, respectively. Moreover, all variables must be initialized before use for the MARC4 core to function normally. The following variables are used in the implementation of Hummingbird:

- `Text0` to `Text3`: 16-bit plaintext/ciphertext block;
- `RSi0` to `RSi3`: 16-bit rotor $RSi$, $i = 1, 2, 3, 4$;
- `LFSR0` to `LFSR3`: 16-stage LFSR;
- `V120` to `V123`: 16-bit output of block cipher $E_{k_1}$;
- `V230` to `V233`: 16-bit output of block cipher $E_{k_2}$;
- `V340` to `V343`: 16-bit output of block cipher $E_{k_3}$;
- `KeySelection`: 4-bit variable for subkey selection.

Besides the above variables, two set of temporary variables `T0` to `T3` and `S0` to `S3` are also defined and initialized.

### 4.2. Internal State Register Initialization

Initializing the internal state registers of Hummingbird can be divided into three phases as shown in Figure 2. These phases are 1) Initializing the rotor $RSi$ with a randomly chosen nonce $\text{NONCE}_i$ ($i = 1, 2, 3, 4$) (see Section 4.1), 2) Encrypting the message $RS1 \boxplus RS3$ using the four block ciphers consecutively and updating the states four rotors for four rounds, 3) Using the final 16-bit ciphertext as the initial state of LFSR with the 13th bit set. The initialization of the internal state registers is implemented by the following qForth word[3] **:SetInitialState** in which the qForth word **:BlockEncrypt** (see Section 4.3.2) implements the 16-bit block cipher encryption.

```
: SetInitialState
  4 #DO                          \ Four rounds encryption and rotor stepping
    [...]
    CLR_BCF                      \ Clear BRANCH and CARRY flag
    V340 @ RS40 @ + Text0 !      \ Compute V34 + RS4
    [...]
    4 KeySelection !             \ Use the 4th 64-bit key k_4
    BlockEncrypt                 \ 16-bit block cipher encryption
    Text0 @ T0 !                 \ Store the ciphertext in T0 to T3
    [...]
    CLR_BCF                      \ Clear BRANCH and CARRY flag
    RS10 @ T0 @ + RS10 !         \ Step rotor RS1(t + 1) = RS1(t) + T
    [...]
  #LOOP
  T0 @ LFSR0 !                   \ Set the initial state of LFSR
  [...]
  T3 @ 0001b OR LFSR3 !          \ Set the 13th bit of LFSR
;
```

[3] A word in qForth is a synonym of a subroutine in other programming languages [2].

### 4.3. Hummingbird Encryption

The encryption of one 16-bit plaintext block $PT_i$ with a given 256-bit secret key $K$ consists of two stages (see Figure 1). $PT_i$ is first encrypted using four block ciphers in a similar manner as the initialization of the internal state registers. Then the states of the four rotors and the LFSR will be updated in terms of the process described in Figure 1. The encryption is implemented by the qForth word **:HummingbirdEnc** which calls two subwords **:BlockEncrypt** (see Section 4.3.2) and **:UpdateLFSR** (see Section 4.3.1).

```
: HummingbirdEnc
  [...]
  CLR_BCF                        \ Clear BRANCH and CARRY flag
  V120 @ RS20 @ + Text0 !        \ Compute V12 + RS2
  [...]
  2 KeySelection !               \ Use the 2nd 64-bit key k_2
  BlockEncrypt                   \ 16-bit block cipher encryption
  Text0 @ V230 !                 \ Store the ciphertext in V230 to V233
  [...]
  CLR_BCF                        \ Clear BRANCH and CARRY flag
  RS30 @ V230 @ + RS30 !         \ Step rotor RS3(t + 1) = RS3(t) + V23 + LFSR
  [...]
  UpdateLFSR                     \ Step the Galois LFSR to the next state
  RS30 @ LFSR0 @ + RS30 !
  [...]
;
```

#### 4.3.1 LFSR and Its State Update

In Hummingbird, a 16-stage LFSR in Galois configuration (see Figure 7) is utilized because it offer minimal latency and higher speed for both software and hardware implementation. In the Galois configuration, when the state of LFSR needs to be updated, bits that are not taps are shifted one position to the right unchanged. The taps, on the other hand, are XOR'd with the output bit before they are stored in the next position. The state update of the Galois LFSR can be efficiently implemented with the qForth word **:UpdateLFSR** as follows:

```
: UpdateLFSR
  LFSR0 @ LFSR1 @ LFSR2 @ LFSR3 @     \ Put LFSR into the stack
  SHR Temp3 !                         \ LFSR >> 1
  [...]
  LFSR0 @ 0001b AND 0001b =           \ Test whether LFSR & 0x1 = 0x1
  IF                                  \ Yes, LFSR = (LFSR >> 1) ^ 0xca44
    [...]
    Temp3 @ 1100b XOR LFSR3 !
  ELSE                                \ No, LFSR = LFSR >> 1
    [...]
    Temp3 @ LFSR3 !
  THEN
;
```

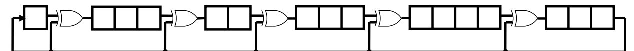

**Figure 7. A 16-bit Galois LFSR with Characteristic Polynomial** $f(x) = x^{16} + x^{15} + x^{12} + x^{10} + x^7 + x^3 + 1$

#### 4.3.2 Block Cipher Encryption

To implement the 16-bit block cipher encryption $E_{k_i}$ as shown in Figure 3, the $4 \times 4$ S-box $S_1$ and the 64-bit subkey

$k_i = K_1^{(i)} \| K_2^{(i)} \| K_3^{(i)} \| K_4^{(i)}$ are first stored in some specific areas of the ROM of the MARC4 core with the qForth command **ROMCONST** as follows:

```
\ 16 nibbles for storing S-BOX used in the Compact Hummingbird
ROMCONST SBOX 8 , 6 , 5 , 15 , 1 , 12 , 10 , 9 ,
             14 , 11 , 2 , 4 , 7 , 0 , 13 , 3 , AT 340h

\ 64-bit subkey k_1 and two derived round keys used in block cipher E_{k_1}
ROMCONST Key1 7eh , 2bh , 16h , 15h , aeh , 28h , a6h , d2h ,
              d0h , 03h , b0h , c7h , AT 360h
[...]
```

Note that two derived round keys $K_5^{(i)} = K_1^{(i)} \oplus K_3^{(i)}$ and $K_6^{(i)} = K_2^{(i)} \oplus K_4^{(i)}$ are also stored in the ROM for efficient implementation. Moreover, the qForth command **ROMByte@** is used to fetch an 8-bit constant from the ROM each time. The four regular rounds of the encryption algorithm can be implemented in a similar way. More specifically, after a simple key mixing step, a message is divided into four 4-bit chunks and each of them is then substituted by a single $4 \times 4$ S-box (see **:DoSbox** below). Subsequently, the 16-bit message is further permuted by the linear transform $L$ (see **:LinearTransform** below). Unlike the regular round, the final round only consists of two key mixing steps with round keys $K_5^{(i)}$ and $K_6^{(i)}$ and an S-box substitution step.

```
: BlockEncrypt
  KeySelection @            \ Select a 64-bit subkey
  CASE
    1 OF
      \ ******Round 1******
      Key1 ROMByte@         \ Fetch the 1st 8-bit of k_1
      Text0 @ XOR Text0 !   \ Add round key
      Text1 @ XOR Text1 !
      [...]
      DoSbox                \ S-box substitution
      LinearTransform       \ Linear transform Text ^ (Text << 6) ^ (Text << 10)
      [...]
    ENDOF
    [...]
  ENDCASE
;
```

The S-box substitution can be efficiently implemented by first pushing the base address (12 bits) of the look-up table into the EXP. The value of the 4-bit chunk is then used as the offset to form the address of the substitution result.

```
: DoSbox
  SBOX Text0 @ +            \ Look-up table addressing
  ROMByte@                  \ S-box substitution
  Text0 !                   \ Store SBOX[Text0] to Text0
  DROP                      \ Drop high nibble which is 0
  [...]
;
```

The linear transform $L$ involves rotating a 16-bit message by 6 and 10 positions to the left, respectively, which is performed by two steps: 1) rotate the message by 4 and 8 positions to the left, respectively; 2) rotate the resulting message by 2 positions to the left. For the MARC4 4-bit architecture, the first step can be easily implemented by re-addressing memory pointers, which is equivalent to change the order that a 16-bit message is pushed into the EXP, as shown in the following qForth word **:LinearTransform**.

```
: LinearTransform
  Text2 @ Text1 @ Text0 @ Text3  \ Push |Text0|Text1|Text2|Text3| onto stack << 4
  ROTL2                          \ T = Text << 6
  Text0 @ T0 @ XOR S0 !          \ S = Text ^ (Text << 6)
  [...]
  Text1 @ Text0 @ Text3 @ Text2  \ Push |Text0|Text1|Text2|Text3| onto stack << 8
  ROTL2                          \ T = Text << 10
  S0 @ T0 @ XOR Text0 !          \ Text = Text ^ (Text << 6) ^ (Text << 10)
  [...]
;
```

The second step is implemented with the qForth word **:ROTL2** (see below), which continuously rotates a 16-bit message by one position to the left twice with the assistance of the MARC4 Conditional Code Register (CCR).

```
: ROTL2
  SHL T3 !                  \ Rotate a 16-bit message by one position to the left
  ROL T2 !                  \ Text3 << 1
  ROL T1 !                  \ Text2 << 1
  ROL T0 !                  \ Text1 << 1
  0111b CCR@ <              \ Test whether there is a carry
  IF T3 @ 1 XOR T3 !        \ If it is, insert 0001b into T3
  ELSE T3 @ 0 XOR T3 !      \ Timing-attack resistance
  THEN
  [...]
;
```

### 4.4. Hummingbird **Decryption**

The decryption procedure of the Hummingbird cryptographic algorithm is quite similar to the encryption process and a top-level description can be found in [5]. From the function call graph in Figure 6, we note that both encryption and decryption share the same qForth words **:UpdateLFSR** and **:ROTL2**, which further reduces the code size and enables us to fit an implementation that is capable of encryption and decryption in spite of the harsh memory constrains of the MARC4 controllers (i.e., 4KB ROM and 128B RAM). Due to page limitations, we omit the description of the implementation of the decryption process. It is not difficult for a reader to implement the decryption algorithm based on the information provided in Section 4.3.

### 4.5. Experimental Results and Comparisons

To the best of our knowledge, the only published implementation of cryptographic algorithms on 4-bit microcontrollers is about the ultra-lightweight block cipher PRESENT [14]. Hence, we compare the performance of the optimized versions of PRESENT and Hummingbird in this subsection. Table 1 summarizes the memory consumption and cycle count of two ciphers on the target 4-bit platform. From Table 1, we note that Hummingbird encryption and decryption algorithms require roughly equal lines of code [4], which is longer than those of PRESENT. The reason is that the loop unrolling technique is employed to optimize the performance of the algorithm in our implementation, which increases the code size significantly. Moreover, both Hummingbird and PRESENT use almost the same number of RAM nibbles as the stack space [5]. In addition, the design of Hummingbird is based on a hybrid mode of block cipher and stream cipher, which results in a relatively long initialization process when compared to the block cipher PRESENT. After the initialization phase, the encryption of a 16-bit plaintext block with Hummingbird

[4] We count the lines of code twice for the qForth words shared by the Hummingbird encryption and decryption algorithms.

[5] While the depth (in RAM nibbles) of the EXP is exactly the number allocated, the RET depth is expressed as $\text{RET}_{\text{value}} = \text{RET}_{\text{depth}} \times 4$.

requires $5,773$ cycles, whereas it takes $55,734$ cycles for PRESENT to encrypt 64-bit plaintext. Furthermore, the decryption of one ciphertext block of $16/64$ bits with Hummingbird/PRESENT requires $5,212$ and $65,574$ cycles on the target 4-bit microcontroller, respectively.

**Table 1. Memory Consumption and Cycle Count of PRESENT and Hummingbird on the ATAM893-D 4-bit Microcontroller**

| enc/ dec | ROM [lines of code] | Stack Depth [EXP/RET] | Init. [cycles] | Cycles/block [cycles] |
|---|---|---|---|---|
| P-enc [14] | 841 | $25/4$ | 230 | $55,734$ |
| P-dec [14] | 945 | $25/4$ | 230 | $65,574$ |
| H-enc | 1532 | $9/7$ | $22,949$ | $5,773$ |
| H-dec | 1559 | $9/7$ | $22,949$ | $5,212$ |

Table 2 estimates the running time and throughput of PRESENT and Hummingbird on the 4-bit ATAM893-D microcontroller clocked at 16KHz, 500KHz and 2MHz, respectively. As one can see from Table 2 that after an initialization phase the Hummingbird encryption and decryption algorithms can achieve nearly $89.6\%$ and $92\%$ faster execution time and $58.6\%$ and $68.2\%$ faster throughput than the state-of-the-art ultra-lightweight block PRESENT at each given clock frequency, respectively.

**Table 2. Timing and Throughput of PRESENT and Hummingbird for Different Clock Frequencies on the ATAM893-D 4-bit Microcontroller**

| enc/ dec | Clock Freq. [KHz] | Clock Source [int./ext.] | Timing [ms] | Throughput [bits/sec] |
|---|---|---|---|---|
| P-enc [14] | $2,000$ | int. | 27.9 | $2,297$ |
| | 500 | ext. | 111.5 | 574 |
| | 16 | ext. | $3,483$ | 18.4 |
| P-dec [14] | $2,000$ | int. | 32.8 | $1,952$ |
| | 500 | ext. | 131.1 | 488 |
| | 16 | ext. | $4,098$ | 15.6 |
| H-enc | $2,000$ | int. | 2.89 | $5,543$ |
| | 500 | ext. | 11.55 | $1,385.8$ |
| | 16 | ext. | 360.8 | 44.3 |
| H-dec | $2,000$ | int. | 2.61 | $6,139.7$ |
| | 500 | ext. | 10.4 | $1,534.9$ |
| | 16 | ext. | 325.8 | 49.1 |

Combining the cost of the system initialization, Table 3 and Table 4 describe the overall performance of PRESENT and Hummingbird for encrypting and decrypting messages with different length on the target 4-bit microcontroller, respectively. From Tables 3 and 4, we note that the longer the message length, the larger performance improvement of Hummingbird over PRESENT. Moreover, for typical RFID applications where most of transmitted messages are usually very short, the advantage of using Hummingbird instead of PRESENT is obvious as well.

**Table 3. The Overall Performance Comparison of PRESENT and Hummingbird Encryption Algorithm for Different Message Length on the ATAM893-D 4-bit Microcontroller**

| Message Length | Clock Freq. [KHz] | P-enc [14] [ms] | H-enc [ms] | Performance Improvement |
|---|---|---|---|---|
| 64-bit | $2,000$ | 28.02 | 23.03 | 17.8% |
| | 500 | 111.96 | 92.1 | |
| | 16 | $3,497.38$ | $2,877.51$ | |
| 128-bit | $2,000$ | 55.92 | 34.59 | 38.1% |
| | 500 | 223.46 | 138.3 | |
| | 16 | $6,980.38$ | $4,320.71$ | |
| 192bit | $2,000$ | 83.82 | 46.15 | 44.9% |
| | 500 | 334.96 | 184.5 | |
| | 16 | $10,463.38$ | $5,763.91$ | |

**Table 4. The Overall Performance Comparison of PRESENT and Hummingbird Decryption Algorithm for Different Message Length on the ATAM893-D 4-bit Microcontroller**

| Message Length | Clock Freq. [KHz] | P-dec [14] [ms] | H-dec [ms] | Performance Improvement |
|---|---|---|---|---|
| 64-bit | $2,000$ | 32.92 | 21.91 | 33.4% |
| | 500 | 131.56 | 87.5 | |
| | 16 | $4,112.38$ | $2,737.51$ | |
| 128-bit | $2,000$ | 65.72 | 32.35 | 50.8% |
| | 500 | 262.66 | 129.1 | |
| | 16 | $8,210.38$ | $4,040.71$ | |
| 192bit | $2,000$ | 98.52 | 42.79 | 56.6% |
| | 500 | 393.76 | 170.7 | |
| | 16 | $12,308.38$ | $5,343.91$ | |

## 5. Conclusions

4-bit microcontrollers have extremely low power consumption (typically 1 – $70\mu A$), making them interesting platforms for implementing various security solutions on both active and passive low-cost RFID tags. In this paper, we describe an efficient implementation and the performance of the ultra-lightweight cryptographic algorithm Hummingbird on the 4-bit ATAM893-D microcontroller of Atmel's MARC4 family. The experimental results show that Hummingbird can achieve better performance than PRESENT on the target platform. Our contribution is another step towards verifying the feasibility of implementing lightweight cryptographic algorithms on ultra-constrained 4-bit microcontrollers for potential RFID applications. As our future work, we would like to propose a side channel attack resistant implementation of Hummingbird on the 4-bit microcontroller. Moreover, we also plan to exploit ultra-low power 4-bit microcontrollers to establish an experimental platform for simulating Hummingbird mutual authentication protocols [5] between an RFID reader and tags.

## References


[1] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard", available at http://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf.

[2] Atmel Corporation, "MARC4 4-bit Microcontrollers Programmer's Guide", available at http://www.atmel.com/dyn/resources/prod_documents/doc4747.pdf.

[3] Atmel Corporation, "ATAM893-D: Flash Version for ATAR080, ATAR090/890 and ATAR092/892", available at http://www.atmel.com/dyn/resources/prod_documents/doc4680.pdf.

[4] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher," *The 9th International Workshop on Cryptographic Hardware and Embedded Systems - CHES 2007*, LNCS 4727, P. Paillier and I. Verbauwhede (eds.), Berlin, Germany: Springer-Verlag, pp. 450-466, 2007.

[5] D. Engels, X. Fan, G. Gong, H. Hu, and E. M. Smith, "Ultra-Lightweight Cryptography for Low-Cost RFID Tags: Hummingbird Algorithm and Protocol," *Centre for Applied Cryptographic Research (CACR) Technical Reports*, CACR 2009-29, available at http://www.cacr.math.uwaterloo.ca/techreports/2009/cacr2009-29.pdf.

[6] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer, "Strong Authentication for RFID Systems Using the AES Algorithm", *The 6th International Workshop on Cryptographic Hardware and Embedded Systems - CHES 2004*, LNCS 3156, M. Joye and J.-J. Quisquater (eds.), Berlin, Germany: Springer-Verlag, pp. 357-370, 2004.

[7] M. Feldhofer, J. Wolkerstorfer, and V. Rijmen, "AES Implementation on a Grain of Sand," *IEE Proceedings Information Security*, vol. 15, no. 1, pp. 13-20, 2005.

[8] P. Hämäläinen, T. Alho, M. Hännikäinen, and T. D. Hämäläinen, "Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core," *The 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools - DSD 2006*, pp. 577-583, IEEE Computer Society, 2006.

[9] ICC Counterfeiting Intelligence Bureau (ed.), "The International Anti-Counterfeiting Directory 2009", available at http://www.icc-ccs.org/images/stories/pdfs/iacd2009.pdf.

[10] G. Leander, C. Paar, A. Poschmann, and K. Schramm, "New Lightweight DES Variants", *The 14th Annual Fast Software Encryption Workshop - FSE 2007*, LNCS 4593, A. Biryukov (ed.), Berlin, Germany: Springer-Verlag, pp. 196-210, 2007.

[11] C. Rolfes, A. Poschmann, G. Leander, and C. Paar, "Ultra-Lightweight Implementations for Smart Devices-Security for 1000 Gate Equivalents", *The 8th Smart Card Research and Advanced Application IFIP Conference - CARDIS 2008*, LNCS 5189, G. Grimaud and F.-X. Standaert (eds.), Berlin, Germany: Springer-Verlag, pp. 89-103, 2008.

[12] T. Staake, F. Thiesse, and E. Fleisch, "Extending the EPC Network: the Potential of RFID in Anti-Counterfeiting", *The 2005 ACM Symposium on Applied Computing - SAC'05*, pp. 1607-1612, 2005.

[13] P. Tuyls and L. Batina, "RFID-tags for Anti-Counterfeiting", *Topics in Cryptology - CT-RSA 2006*, LNCS 3860, D. Pointcheval (ed.), Berlin, Germany: Springer-Verlag, pp. 115-131, 2006.

[14] M. Vogt, A. Poschmann, and C. Paar, "Cryptography is Feasible on 4-Bit Microcontrollers - A Proof of Concept", *2009 IEEE International Conference on RFID*, pp. 241-248, 2009.