High Speed Implementation of Authenticated Encryption for the MSP430X Microcontroller

Conrado P. L. Gouvêa[⋆], Julio López

University of Campinas (Unicamp),
{conradoplg,jlopez}@ic.unicamp.br

Abstract. Authenticated encryption is a symmetric cryptography algorithm that provides both confidentiality and authentication in a single scheme. In this work we describe an optimized implementation of authenticated encryption for the MSP430X family of microcontrollers. The CCM, GCM, SGCM, OCB3 and Hummingbird-2 modes of authenticated encryption were implemented in the 128-bit level of security and their performance was compared. The AES accelerator present in some models of the MSP430X family is also studied and we explore its characteristics to improve the performance of the implemented modes, achieving up to 13 times of speedup.

1 Introduction

Constrained platforms such as sensor nodes, smart cards and radio-frequency identification (RFID) devices have a great number of applications, many of which with security requirements that require cryptographic schemes. The implementation of such schemes in these devices is very challenging since it must provide high speed while consuming a small amount of resources (energy, code size and RAM).

In this scenario, symmetric cryptography becomes an essential tool in the development of security solutions, since it can provide both confidentiality and authenticity after being bootstrapped by some protocol for key agreement or distribution (for example, with public key cryptography using elliptic curves [7], or identity-based cryptography using pairings [13]). Encryption and authentication can be done through generic composition of separate methods; however, the study of an alternative approach named authenticated encryption (AE) has gained popularity.

Authenticated encryption provides both confidentiality and authenticity within a single algorithm. It is often more efficient than using separate methods and usually consumes a smaller amount of resources. It also prevents common critical mistakes when combining encryption and authentication such as not using separate keys for each task, applying the methods in the wrong order (encrypt-then-authenticate is the only order proven to work with any underlying secure schemes [1]), or not authenticating the initialization vector used for encryption.

[⋆] Supported by FAPESP, grant 2010/15340-3.

There are many AE modes; see e.g. [9] for a non-exhaustive list. In this work, we follow the approach from [9] and compare the Counter with CBC-MAC (CCM) mode [20], the Galois/Counter Mode (GCM) [12] and the Offset Codebook (OCB3) mode [9]. We have also implemented the Sophie Germain Counter Mode [16] and the Hummingbird-2 cipher [4]. The CCM mode and GCM have been standardized by the National Institute of Standards and Technology (NIST); CCM is used for Wi-Fi WPA2 security (IEEE 802.11i) while GCM is used in TLS, IPSec and NSA Suite B, for example. The recently proposed OCB3 mode is the third iteration of the OCB mode and appears to be very efficient in multiple platforms. The SGCM is a variant of GCM and was proposed to be resistant against some existing attacks against GCM while being equally or more efficient; we have implemented it in order to check this claim and compare it to GCM. The Hummingbird-2 (which may be referred to as HB2 in this work) is specially suited for 16-bit platforms and was implemented in order to compare it to the other non-specially suited modes.

The main goal of this work is to provide an efficient implementation and comparison of the aforementioned AE modes (CCM, GCM, SGCM, OCB3 and Hummingbird-2) for the MSP430X microcontroller family from Texas Instruments. This family is an extension of the MSP430 which have been used in multiple scenarios such as wireless sensor networks [8,19]; furthermore, some microcontrollers of this family feature an AES accelerator module which can encrypt and decrypt using 128-bit keys. Our contributions are: to study (for the first time, to the best of our knowledge) the efficient usage and impact of this AES accelerator module in the implemented AE modes; to describe a high speed implementation of those AE modes for the MSP430X, achieving performance more than 13 times faster for CCM using the AES accelerator instead of AES in software; to show that CCM is the fastest of those modes whenever a non-parallel AES accelerator is available; and to provide a comparison of the five AE modes, with and without the AES accelerator. We remark that the results regarding the efficient usage of the AES accelerator can probably be applied to other devices featuring analogue accelerators, such as the AVR XMEGA.

This paper is organized as follows. In Section 2, the MSP430X microcontroller family is described. Section 3 offers an introduction to AE, along with a description and comparison of the implemented modes. Our implementation is described in Section 4, and the obtained results are detailed in Section 5. Section 6 provides concluding remarks.

2 The MSP430X Family

The MSP430X family is composed by many microcontrollers which share the same instruction set and 12 general purpose registers. Although it is essentially a 16-bit architecture, its registers have 20 bits, supporting up to 1 MB of addressing space. Each microcontroller model has distinct clock frequency, RAM and ROM (flash) size.

Some MSP430X microcontrollers (namely the CC430 series) have an integrated radio frequency transceiver, making them very suitable for wireless sensors. These models also feature an AES accelerator module that supports encryption and decryption with 128-bit keys only. The study of this accelerator is one key aspect of this study and for this reason we describe its basic usage as follows. In order to encrypt a block of 16 bytes, a flag must be set in a control register to specify encryption and the key must be written sequentially (in bytes or words) in a specific memory address. The input block must then be written, also sequentially, in another memory address. After 167 clock cycles, the result is ready and must be read sequentially from a third address. It is possible to poll a control register to check if the result is ready. Further blocks can be encrypted with the same key without writing the key again. The decryption follows the same procedure, but it requires 214 clock cycles of processing. It is worth noting that these memory read and writes are just like regular reads and writes to the RAM, and take the same time to be performed.

3 Authenticated Encryption Modes

An authenticated encryption mode is composed of two algorithms: authenticated encryption and decryption-verification (of integrity). The authenticated encryption algorithm is denoted by the function $\mathcal{E}_K(N, M, A)$ that returns (C, T), where $K \in \{0,1\}^k$ is the k-bit key, $N \in \{0,1\}^n$ is the n-bit nonce, $M \in \{0,1\}^*$ is the message, $A \in \{0,1\}^*$ is the associated data, $C \in \{0,1\}^*$ is the ciphertext and $T \in \{0,1\}^t$ is the authentication tag. The nonce is a non-secret value that must be unique for each message and prevents the same plaintext being always encrypted to the same ciphertext. (Some modes support variable-length nonces, but we have fixed its size to simplify the exposition. The same applies to the tag.) The associated data (AD) is authenticated by the algorithm, but not encrypted; this can be useful if some header must be sent in plaintext along with the encrypted message, for example, in an internet packet. The decryption-verification algorithm is denoted by the function $\mathcal{D}_K(N, C, A, T)$ that returns (M, V) where K, N, C, A, T, M are as above and V is a boolean value indicating if the given tag is valid (i.e. if the decrypted message and associated data are authentic).

Most AE modes are built using a block cipher such as AES. Let $E_K(B)$ denote the block cipher, where the key K is usually the same used in the AE mode and $B \in \{0,1\}^b$ is a b-bit message (a *block*). The inverse (decryption) function is denoted $D_K(B)$.

It is possible to identify several properties of AE modes; we offer a non-exhaustive list. The *number of block cipher calls* used in the mode is an important metric related to performance. A mode is considered *online* if it is able to encrypt a message with unknown length using constant memory (this is useful, for example, if the end of the data is indicated by a null terminator or a special packet). Some modes *only use the forward function* of the underlying block cipher (E_K), which reduces the size of software and hardware implementations. A mode *supports preprocessing of static AD* if the authentication of the AD depends only on the key and can be cached between different messages being sent (this is useful for a header that does not change). Some modes are *covered by patents*, which usually discourages its use. A mode is *parallelizable* if it is possible to process multiple blocks (or partially process them) in a parallel manner. Some modes support *processing regular messages and AD in any order*, while some modes require the processing of AD before the message, for example. The properties of the AE modes implemented in this work are compared in Table 1.

The following notation is used in the description of the algorithms. Let $A \oplus B$ denote the logical xor of the bit strings A and B (if the strings have different sizes, align them to left and discard the excess from the larger string) and let $A \mid B$ denote the logical *or* in the same fashion. Let $A[i..j]$ denote the substring of the bit string A starting in the i-th bit and ending in the j-th bit, inclusive. The same slicing notation is used for array of words in Algorithm 4. Let $[i]_n$ denote the n-bit representation of the integer i (endianess will be made explicit in the description of the algorithm). Write $A \mid\mid B$ for the concatenation of the bit strings A and B. Write 0^{128} for the block of 128 bits filled with zeros. The AE algorithms will be presented in a simplified manner, omitting details in the handling of incomplete blocks (i.e. with size smaller than the block size) and the decryption-verification algorithms. We refer the reader to the original papers for their complete description.

Table 1. Comparison of implemented AE modes

Property	CCM	(S)GCM	OCB3	HB2
Block cipher calls*	$2m + a + 2^\dagger$	m	$m + a + 1^\dagger$	—
... in key setup	0	1	1	—
Online	No	Yes	Yes	Yes‡
Uses only E_K	Yes	Yes	No	—
Preprocessing of static AD	No	Yes	Yes	No
Patent-free	Yes	Yes	No	No
Parallelizable	No	Yes	Yes	No
Standardized	Yes	(No) Yes	No	No
Order of message and AD	AD first	AD first	Any	AD last

* m, a are the number of message and AD blocks, respectively
† May have an additional block cipher call
‡ AD size must be fixed

3.1 CCM

The CCM (Counter with CBC-MAC) mode [20] essentially combines the CTR mode of encryption with the CBC-MAC authentication scheme. It requires two cipher block calls for each block to be encrypted. Algorithm 1 presents CCM, where the function `format` computes a header block B_0 (which encodes the tag length, message length and nonce), the blocks $A_1, \ldots, A_a$ (which encode the length of the associated data along with the data itself) and the blocks $M_1, \ldots, M_m$ which represent the original message. The function `init_ctr` returns the initial counter based on the nonce. The function `inc` increments the counter.

Properties. CCM is not online since the message length is encoded in the header block, which is the first to be processed. It is not very well parallelizable since the authentication requires the result of the previous block cipher call in order to authenticate the current block. It is not possible to preprocess static AD since the authentication depends on B_0, which is based on the nonce. CCM has also been criticized for disrupting word alignment (among other reasons [14]), since the header attached to the AD can have 2, 6 or 10 bytes (this also requires copying chunks of the AD to a buffer before xoring it and sending it to the block cipher).

Endianess issues. The encoding of the lengths and counters must be in big endian format; otherwise, CCM is not affected by endianess issues.

Algorithm 1 CCM encryption

```
Input: Message $M$, additional data $A$, nonce $N$, key $K$
Output: Ciphertext $C$, authentication tag $T$ with $t$ bits
 1: $B_0, A_1, \ldots, A_a, M_1, \ldots, M_m \leftarrow \texttt{format}(N, A, M)$
 2: $Y \leftarrow E_K(B_0)$
 3: for $i \leftarrow 1$ to $a$ do
 4:     $Y \leftarrow E_K(A_i \oplus Y)$
 5: end for
 6: $J \leftarrow \texttt{init\_ctr}(N)$
 7: $S_0 \leftarrow E_K(J)$
 8: $J \leftarrow \texttt{inc}(J)$
 9: for $i \leftarrow 1$ to $m$ do
10:     $U \leftarrow E_K(J)$
11:     $J \leftarrow \texttt{inc}(J)$ {delay slot}
12:     $S \leftarrow M_i \oplus Y$ {delay slot}
13:     $Y \leftarrow E_K(S)$
14:     $C_i \leftarrow M_i \oplus U$ {delay slot}
15: end for
16: $T \leftarrow Y[0..t-1] \oplus S_0[0..t-1]$
```

3.2 GCM

The GCM (Galois/Counter Mode) [12] employs the arithmetic of the finite field (Galois field) $\mathbb{F}_{2^{128}}$ for authentication and the CTR mode for encryption. It requires a single block cipher call for each block to be encrypted. Algorithm 2 describes GCM, where the function `init_ctr` initializes the counter and the function `inc_ctr` increments the counter. The operation $A \cdot B$ denotes the multiplication of A and B in $\mathbb{F}_{2^{128}}$. The mode benefits from precomputed lookup tables since the second operand is fixed for all multiplications (lines 6, 15 and 18 from Algorithm 1).

Properties. GCM is online and parallelizable. In fact, it has most of the “good” properties from Table 1. It is possible to employ different sizes for the precomputation lookup table as a speed/space tradeoff, varying from 256 bytes to 64 KB.

Endianess issues. In order to interpret a string of bytes as a $\mathbb{F}_{2^{128}}$ element, GCM chooses to view the bytes in a little endian fashion. More peculiarly, it treats the bits inside a byte in a reversed manner: the first bit (i.e. obtained with `c & 1` in the C language) is the *most* significant bit. Therefore, the element $a(z) = 1$ is represented as the byte string `80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`.

Algorithm 2 GCM encryption

```
Input: Message $M$, additional data $A$, nonce $N$, key $K$
Output: Ciphertext $C$, authentication tag $T$ with $t$ bits
 1: $A_1, \ldots, A_a \leftarrow A$
 2: $M_1, \ldots, M_m \leftarrow M$
 3: $H \leftarrow E_K(0^{128})$
 4: $Y \leftarrow 0^{128}$
 5: for $i \leftarrow 1$ to $a$ do
 6:     $Y \leftarrow (A_i \oplus Y) \cdot H$
 7: end for
 8: $J \leftarrow \texttt{init\_ctr}(N)$
 9: $S_0 \leftarrow E_K(J)$
10: $J \leftarrow \texttt{inc}(J)$
11: for $i \leftarrow 1$ to $m$ do
12:     $U \leftarrow E_K(J)$
13:     $J \leftarrow \texttt{inc}(J)$ {delay slot}
14:     $C_i \leftarrow M_i \oplus U$
15:     $Y \leftarrow (C_i \oplus Y) \cdot H$
16: end for
17: $L \leftarrow [len(A)]_{64} \,||\, [len(M)]_{64}$
18: $S \leftarrow (L \oplus Y) \cdot H$
19: $T \leftarrow (S \oplus S_0)[0..t-1]$
```

3.3 SGCM

The SGCM (Sophie Germain Counter Mode) [16] is a variant of GCM that is not susceptible to weak key attacks that exist against GCM. While these attacks are of limited nature, the author claims that they should be avoided. It has the same structure as GCM, but instead of the $\mathbb{F}_{2^{128}}$ arithmetic, it uses the prime field $\mathbb{F}_p$ with $p = 2^{128} + 12451$.

Properties. The same as GCM.

Endianess issues. Elements of $\mathbb{F}_p$ are represented by little-endian byte arrays. However, unlike GCM, the bits inside a byte are viewed in the usual manner: the first bit (i.e. obtained with `c & 1` in the C language) is the least significant bit. Therefore, the element 1 is represented as the byte string `01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`.

3.4 OCB3

The OCB3 (Offset Codebook) mode [9] also employs the $\mathbb{F}_{2^{128}}$ arithmetic (using the same reduction polynomial from GCM), but in a simplified manner: it does not require full multiplication, but only multiplication by powers of z (the variable used in the polynomial representation of the field elements). It also requires a single block cipher call for each block being encrypted. It is described in Algorithm 3, where the function `init_delta` derives a value from the nonce and it may require a block cipher call, as explained later. The function `ntz`(i) returns the number of trailing zeros in the binary representation of i (e.g. `ntz`$(1) = 0$, `ntz`$(2) = 1$). The function `getL(`L_0`,` x`)` computes the field element $L_0 \cdot z^x$ and can benefit from a precomputed lookup table. Notice that the multiplication by z is simply a left shift of the operand by one bit, discarding the last bit and xoring the last byte of the result with 135 (which is the representation of $z^7+z^2+z^1+1$) if the discarded bit was 1. The function `hash` authenticates the additional data and is omitted for brevity.

Properties. OCB3 is also online and parallelizable, but uses both E_K and D_K and is covered by patents [15]. There is an interesting feature in the `init_delta` function: it requires a block cipher call whose input is the nonce, padded to the left with zeros to fill a block, and with the lower 6 bits set to zero. Therefore, when the nonce is a counter (which is often the case), the block cipher result can be cached between messages, saving a block cipher call 98% of the time.

Endianess issues. In order to interpret a string of bytes as a $\mathbb{F}_{2^{128}}$ element, OCB3 chooses to view the bytes in a big endian fashion. The bits inside a byte are viewed in the usual manner: the first bit (`c & 1`) is the least significant bit. Therefore, the field element $a(z) = 1$ is represented as the byte string `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01`.

3.5 Hummingbird-2 (HB2)

The Hummingbird-2 [4] is an authenticated encryption algorithm which is not built upon a block cipher. It processes 16-bit blocks and was specially designed

Algorithm 3 OCB3 mode encryption

Input: Message M, additional data A, nonce N, key K
Output: Ciphertext C, authentication tag T with t bits

```
1: $A_1, \ldots, A_a \leftarrow A$
2: $M_1, \ldots, M_m \leftarrow M$
3: $L_* \leftarrow E_K(0^{128})$
4: $L_\$ \leftarrow L_* \cdot z$
5: $L_0 \leftarrow L_\$ \cdot z$
6: $Y \leftarrow 0^{128}$
7: $\mathit{\Delta} \leftarrow \texttt{init\_delta}(N, K)$
8: for $i \leftarrow 1$ to $m$ do
9:     $\mathit{\Delta} \leftarrow \mathit{\Delta} \oplus \texttt{getL}(L_0, \texttt{ntz}(i))$
10:    $U \leftarrow E_K(M_i \oplus \mathit{\Delta})$
11:    $Y \leftarrow Y \oplus M_i$ {delay slot}
12:    $C_i \leftarrow U \oplus \mathit{\Delta}$
13: end for
14: $\mathit{\Delta} \leftarrow \mathit{\Delta} \oplus L_\$$
15: $F \leftarrow E_K(Y \oplus \mathit{\Delta})$
16: $G \leftarrow \texttt{hash}(K, A)$
17: $T \leftarrow (F \oplus G)[0..t-1]$
```

for resource-constrained platforms. The small block size is achieved by maintaining an 128-bit internal state that is updated with each block processed. Authenticated data is processed after the confidential data by simply processing the blocks and discarding the ciphertext generated. The algorithm is built upon the following functions for encryption:

$$\begin{aligned}
S(x) &= S_4(x[0..3]) \mid (S_3(x[4..7]) \ll 4) \\
&\quad \mid (S_2(x[8..11]) \ll 8) \mid (S_1(x[12..15]) \ll 12) \\
L(x) &= x \oplus (x \lll 6) \oplus (x \lll 10) \\
f(x) &= L(S(x)) \\
\texttt{WD16}(x,a,b,c,d) &= f(f(f(f(x \oplus a) \oplus b) \oplus c) \oplus d)\,;
\end{aligned}$$

and their inverses for decryption:

$$\begin{aligned}
S^{-1}(x) &= S_4^{-1}(x[0..3]) \mid (S_3^{-1}(x[4..7]) \ll 4) \\
&\quad \mid (S_2^{-1}(x[8..11]) \ll 8) \mid (S_1^{-1}(x[12..15]) \ll 12) \\
L^{-1}(x) &= x \oplus (x \lll 2) \oplus (x \lll 4) \oplus (x \lll 12) \oplus (x \lll 14) \\
f^{-1}(x) &= S^{-1}(L^{-1}(x)) \\
\texttt{WD16}^{-1}(x,a,b,c,d) &= f^{-1}(f^{-1}(f^{-1}(f^{-1}(x) \oplus d) \oplus c) \oplus b) \oplus a\,;
\end{aligned}$$

where S_1, S_2, S_3, S_4 are S-boxes and $\lll$ denotes the circular left shift of a 16-bit word. The function `WD16` is called four times for each block, therefore f is called 16 times for each block. We refer to [4] for further details.

Properties. Hummingbird-2 is patented. The AD must have fixed size, and must be processed after the confidential data.

Endianess issues. The algorithm uses the addition of 16-bit words, which must be viewed in little-endian fashion.

4 Efficient Implementation

We have written a fast software implementation of the AE modes in the C language, with critical functions written in assembly. The target chip was a CC430F6137 with 20 MHz clock, 32 KB flash for code and 4 KB RAM. The compiler used was the IAR Embedded Workbench version 5.20.

The interface to the AES accelerator was written in assembly, along with a function to xor two blocks and another to increment a block. In order to speed up the GCM mode, polynomial multiplication was implemented in unrolled assembly with the López-Dahab (LD) [11] algorithm using 4-bit window and two lookup tables, as described in Algorithm 4. The first precomputation lookup table holds the product of H and all 4-bit polynomials. Each of the 16 lines of the table has 132 bits, which take 9 words. This leads to a table with 288 bytes. The additional lookup table (which can be computed from the first one, shifting each line 4 bits to the left) allows the switch from three 4-bit shifts of 256-bit blocks to a single 8-bit shift of a 256-bit block, which can be computed efficiently with the `swpb` (swap bytes) instruction of the MSP430.

For SGCM, arithmetic in $\mathbb{F}_p$ can be carried with known algorithms such as Comba multiplication. We follow the approach in [6] which takes advantage of the multiply-and-accumulate operation present in the hardware multiplier of the MSP430 family. However, it must be noted that the CC430 series has a 32-bit multiplier; we have used it to improve even more the performance of the multiplication. The special form of the prime p allows fast modular reduction by taking advantage of the congruence $2^{128}x \equiv -12451 \pmod{p}$ — that is, to reduce a 256-bit value module p, multiply the higher 128 bits by -12451 and add it to the lower 128 bits, repeating until a 128-bit value is obtained.

In the OCB3 mode, a lookup table with 8 entries (128 bytes) was used to speed up the `getL` function. Two functions were implemented in assembly: doubling (using left shifts) and the `ntz` function (using right shifts).

For Hummingbird-2, we have unrolled the `WD16` function. The function f is critical since it is called 16 times per block and must be very efficient; our approach is to use two precomputed lookup tables f_L, f_H each one with 256 2-byte elements, such that $f(x) = f_L[x \mathbin{\&} \texttt{0xFF}] \oplus f_H[(x \mathbin{\&} \texttt{0xFF00}) \gg 8]$. These tables are generated by computing $f_L[x] \leftarrow L(S_4(x[0..3]) \mid (S_3(x[4..7]) \ll 4))$ for every byte x and $f_H[x] \leftarrow L((S_2(x[8..11]) \ll 8) \mid (S_1(x[12..15]) \ll 12))$ also for every byte x. This optimization does not apply for $f^{-1}(x)$ since the inverse S-boxes are applied after the shifts in $L^{-1}(x)$ (this is the reason why decryption is slower than encryption, as will be shown). In this case, we have used precomputed lookup tables L_L, L_H such that $L(x) = L_L[x \mathbin{\&} \texttt{0xFF}] \oplus L_H[(x \mathbin{\&} \texttt{0xFF00}) \gg 8]$.

These are computed as $f_L[x] \leftarrow L(x[0..7]), f_H[x] \leftarrow L(x[8..15] \ll 8)$ for every byte x. The four 4-bit inverse S-boxes have been merged in two 8-bit inverse S-boxes S_L^{-1}, S_H^{-1} such that $S^{-1}(x) = S_L^{-1}(x[0..7]) \mid (S_H^{-1}(x[8..15]) \ll 8)$.

In order to perform comparisons, we have used a software implementation of AES from [5] (the byte-oriented version, with the `VERSION_1` option disabled). It uses approximately 2 KB of precomputed lookup tables to improve speed.

Algorithm 4 López-Dahab multiplication in $\mathbb{F}_{2^{128}}$ for 16-bit words and 4-bit window, using 2 lookup tables.

```
Input: $a(z) = a[0..7], b(z) = b[0..7]$
Output: $c(z) = c[0..15]$
 1: Compute $T_0(u) = u(z)b(z)$ for all polynomials $u(z)$ of degree lower than 4.
 2: Compute $T_1(u) = u(z)b(z)z^4$ for all polynomials $u(z)$ of degree lower than 4.
 3: $c[0..15] \leftarrow 0$
 4: for $k \leftarrow 1$ down to $0$ do
 5:     for $i \leftarrow 0$ to $7$ do
 6:         $u_0 \leftarrow (a[i] \gg (8k)) \bmod 2^4$
 7:         $u_1 \leftarrow (a[i] \gg (8k+4)) \bmod 2^4$
 8:         for $j \leftarrow 0$ to $8$ do
 9:             $c[i+j] \leftarrow c[i+j] \oplus T_0(u_0)[j] \oplus T_1(u_1)[j]$
10:         end for
11:     end for
12:     if $k > 0$ then
13:         $c(z) \leftarrow c(z)z^8$
14:     end if
15: end for
16: return $c$
```

4.1 Using the AES accelerator

As previously mentioned, the AES encryption and decryption using the AES hardware accelerator requires waiting for 167 and 214 cycles, respectively, before reading the results. The key to a efficient implementation using the module is to use this "delay slot" to carry other operations that do not depend on the result of the encryption/decryption. In the listed algorithms, we have ordered the steps as they were implemented in order to take advantage of the delay slot. Take CCM as example (Algorithm 1). In line 10, J is written as input to the AES accelerator. Then, lines 11 and 12 can be executed in the delay slot. Finally, before line 13, $E_K(J)$ can be read from the AES accelerator (waiting until it is ready).

In CCM, the first delay slot is used to compute an increment and a xor (lines 11 and 12 in Algorithm 1) and the second delay slot is used for a xor (line 14 in Algorithm 1). In GCM, the increment in the line 13 of Algorithm 2 is computed in the delay slot. In OCB3, the delay slot is used to compute the xor in the line 3 of Algorithm 3.

5 Results

The performance of the implemented AE modes was measured for the authenticated encryption and decryption-verification of messages with 16 bytes and 4 KB, along with the Internet Performance Index (IPI) [12], which is a weighted timing for messages with 44 bytes (5%), 552 bytes (15%), 576 bytes (20%), and 1500 bytes (60%). For each message size, we have measured the time to compute all nonce-dependent values along with time for authenticated encryption / decryption-verification with 128-bit tags. The derivation of key-dependent values is not included. For OCB3, it was assumed that the block cipher call in `init_ctr` was cached.

The timings were obtained using a development board with a CC430F6137 chip and are reported on Table 2. The number of cycles taken by the algorithms was measured using the built-in cycle counter present in the CC430 models, which can be read in the IAR debugger. Stack usage was also measured using the debugger. Code size was determined from the reports produced by the compiler, adding the size for text (code) and constants.

Table 2. Timings of implemented AE modes for different message lengths, in cycles per byte

	Using AES accelerator			Using AES in software*		
Mode	16 bytes	IPI	4 KB	16 bytes	IPI	4 KB
Encryption						
CTR†	26	23	23	248	246	245
CCM	131	38	36	1 600	493	479
GCM	426	183	180	863	403	396
SGCM	242	89	87	674	306	301
OCB3	144	39	38	621	261	257
HB2‡				569	200	196
Decryption						
CTR†	26	23	23	248	246	245
CCM	144	47	46	1 603	493	479
GCM	429	183	180	862	404	397
SGCM	243	89	87	675	307	302
OCB3	217	48	46	749	385	382
HB2‡				669	297	292

* Based on the software AES implementation from [5]
† Non-authenticated encryption mode included for comparison
‡ It does not use AES

Using the AES accelerator. First, we analyze the results using the AES accelerator, for IPI and 4 KB messages. The GCM performance is more than 5 times slower than the other modes; this is due to the complexity of the full binary field multiplication. The SGCM is more than 50% faster than GCM, since the prime field arithmetic is much faster on this platform, specially using the 32-bit hardware multiplier. Still, it is slower than the other modes. Both CCM and OCB3 have almost the same speed, with CCM being around 5% faster. This is surprising, since that OCB3 essentially outperforms CCM in many platforms [9]. It is explained by the combination of two facts: the hardware support for AES, which reduces the overhead of an extra block cipher call in CCM; and the fact that the AES accelerator does not support parallelism, which prevents OCB3 from taking advantage of its support for it. We have measured that the delay slot optimization improves the encryption speed of GCM, SGCM and OCB3 by around 12% and CCM by around 24%.

Using the AES in software. We now consider the performance using the software AES implementation, for large messages. For reference, the block cipher takes 231 cycles per byte to encrypt and 356 cycles per byte to decrypt. The CCM mode becomes slower due to the larger overhead of the extra block cipher call. The GCM is still slower than OCB3 due to its expensive field multiplication. The SGCM is also faster than GCM, but the improvement is diluted to 25% with the software AES. The Hummingbird-2 cipher outperforms the other modes and is the most efficient if the AES accelerator is not available.

AES accelerator vs. AES in software. Using the AES accelerator, it is possible to encrypt in the CTR mode approximately 10.5 times faster than using AES in software; and it is possible to encrypt with CCM approximately 13 times faster for encryption and 10.6 times faster for decryption. The AES accelerator speedup for GCM and OCB3 is smaller (around 2.2 and 6.6, respectively), due to the larger software overhead of both.

Encryption vs. decryption. When considering the usage of the AES accelerator, GCM has roughly the same performance in encryption and decryption, since the algorithm for both is almost equal; the same applies for SGCM. For both CCM and OCB3, decryption is around 25% and 20% slower, respectively. This is explained by the differences in the data dependencies of the decryption, which prevents the useful use of the delay slot, and that D_K (used by OCB3) is slower than E_K in the AES accelerator. Considering now the usage of the AES in software, encryption and decryption have the same performance in CCM and GCM (since there is no delay slot now) but decryption is almost 50% slower for OCB3, since the underlying block cipher decryption is also slower than the encryption. This results in the OCB3 decryption being slower than SGCM. The decryption in Hummingbird-2 is almost 50% slower due to the $f^{-1}(x)$ function not being able to be fully precomputed, in contrast to $f(x)$. It is interesting to note that the decryption timings are often omitted in the literature, even though they may be substantially different from the encryption timings.

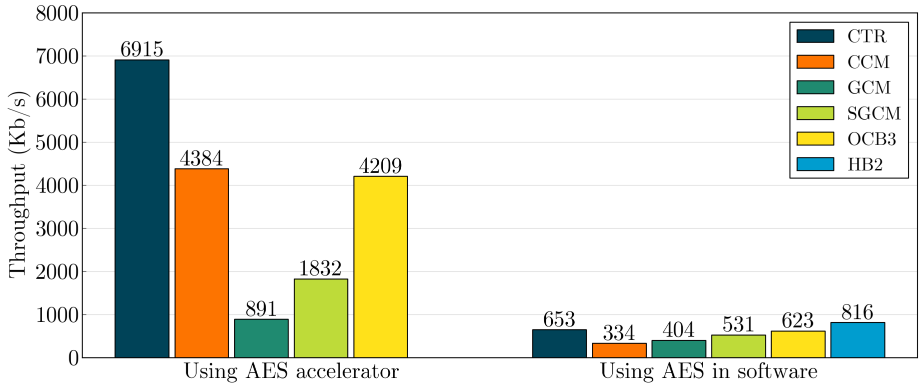

Fig. 1. Encryption throughput in Kbps of CTR and AE modes for 4 KB messages at 20 MHz

Performance for small messages. The timings for 16-byte messages are usually dominated by the computation of nonce-dependent values. The CCM has the worst performance using software AES since all of its initialization is nonce-dependent (almost nothing is exclusively key-dependent) and it includes two block cipher calls. When using the AES accelerator, this overhead mostly vanishes. The nonce setup of GCM is very cheap (just a padding of the nonce) while the nonce setup of OCB3 requires the left shift of an 192-bit block by 0–63 bits. Still, the GCM performance for 16-byte messages is worse than OCB3 since it is still dominated by the block processing. Comparing with the other modes using software AES, the Hummingbird-2 cipher is the fastest for small messages due to its small block size. It is also worth mentioning that, when using smaller tags, Hummingbird-2 is even faster since its tag is generated in 16-bit words at a time while the other algorithms generate a 128-bit tag which can then be truncated.

Further analysis. Figures 1 and 2 present the throughput of encryption and decryption in the CTR and AE modes, considering the 20 MHz clock of the CC430F6137. For comparison, consider the AES software implementation from [3] (also based on [5]) which achieved 286 Kbps at 8 MHz in the ECB mode. Scaling this to 20 MHz we get 716 Kbps, while our ECB implementation achieved 691 Kbps. This is 3.5% slower (probably since we have not spent much time fine-tuning it), but is good enough for our purposes (comparing the performance to the AES accelerator). In [3], it is also claimed that since the maximum throughput of the transceiver is 250 Kbps, it is not needed to encrypt faster than this. This may be true, but a faster (authenticated) encryption uses less energy and may free the controller for other data processing.

Table 3 lists the ROM and RAM usage for programs implementing AE modes for both encryption and decryption, using the AES accelerator. We recall that the MSP430X model we have used features 32 KB of flash for code and 4 KB RAM. The code for GCM is the largest due to the unrolled $\mathbb{F}_{2^{128}}$ multiplier, while

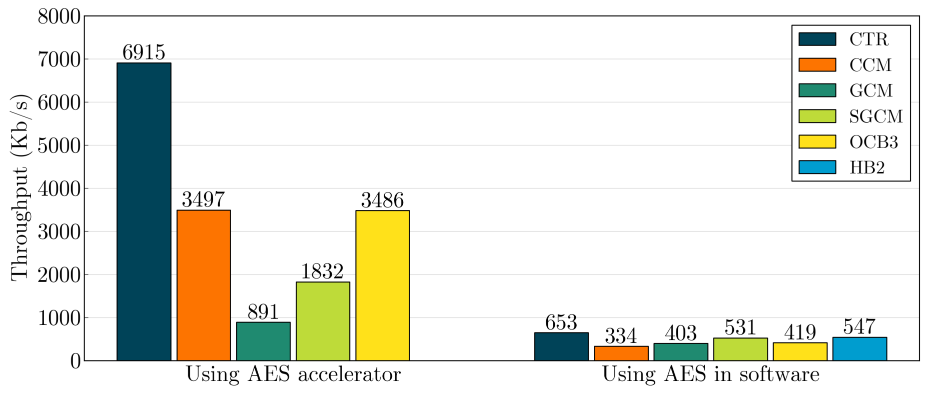

Fig. 2. Decryption throughput in Kbps of CTR and AE modes for 4 KB messages at 20 MHz

the code for CCM is the smallest since it mostly relies on the block cipher. The RAM usage follows the same pattern: GCM has the largest usage, since it has the largest precomputation table; the Hummingbird-2 cipher (followed by CCM) has the smallest RAM usage since it requires no runtime precomputation at all. When using the software AES implementation, 1 564 additional ROM bytes are required for CCM, GCM and SGCM (which use E_K only) and 3 668 additional ROM bytes are required for OCB3.

Table 3. ROM and RAM (stack) usage of AE modes, in bytes. When using software AES, 1 564 additional ROM bytes are required for CCM, GCM and SGCM and 3 668 bytes for OCB3

Mode	ROM	RAM
CTR	688	124
CCM	1 684	250
GCM	5 602	884
SGCM	2 874	322
OCB3	2 382	538
HB2	3 674	196

5.1 Related work

Unfortunately, we are not aware of any works describing implementations of these modes for the MSP430X (except Hummingbird-2), which prevents further comparisons. However, it is still possible to draw some comparisons with related works, as follows.

In [3], the encryption performance using the AES module present in the CC2420 transceiver is studied, achieving 110 cycles per byte. This is still 5 times slower than our results for the CTR mode, probably because the CC2420 is a peripheral and communicating with it is more expensive.

The Dragon-MAC [10] is based on the Dragon stream cipher. Its authors describe an implementation for the MSP430 that achieves 21.4 cycles per byte for authenticated encryption (applying Dragon then Dragon-MAC), which is faster than all timings in this work. However, it requires 18.9 KB of code. Our CCM implementation using the AES accelerator is 1.7 times slower, but 11 times smaller.

The Hummingbird-2 timings reported for the MSP430 in its paper [4] are about 10% faster than the timings we have obtained. However, its authors do not describe their optimization techniques, nor the exact MSP430 model used and their timing methodology, making it difficult to explain their achieved speed. However, we believe that our implementation is good enough for comparisons.

The work [9], whose approach we have followed, provides timings for CTR, CCM, GCM and OCB3 for many platforms, but not for the MSP430. In order to make a comparison, consider the performance relative to the CTR timings. For the x86-64 platform with AES New Instructions, they obtain a performance of 3.28, 2.94 and 1.16 for CCM, GCM and OCB3 respectively; while our results using the AES accelerator are 1.62, 7.87 and 1.71 (notice that our CCM is much faster); using software AES, they are 2.01, 1.64, 1.06 (notice the same ordering of performance).

6 Conclusion

Authenticated encryption modes are a very useful tool in the development of security solutions for constrained platforms. In this work, we have presented an efficient implementation for the MSP430X family of microcontrollers of two popular and standardized AE modes, CCM and GCM, along with the SGCM and OCB3 modes and the Hummingbird-2 cipher. We have also described how to take full advantage of the AES accelerator present in some MSP430X models, achieving a speedup of around 10 times for CTR encryption and CCM compared to the best known timings for a software implementation.

The CCM and OCB3 modes were found to provide similar speed results using the AES accelerator, with CCM being around 5% faster. While OCB3 is the fastest mode in many platforms, we expect CCM to be faster whenever a non-parallel AES accelerator is available. This is the case for the MSP430X models studied and is also the case for other platforms, for example, the AVR XMEGA microcontroller with has an AES module analogue to the MSP430X AES accelerator.

The CCM appears to be the best choice for MSP430X models with AES accelerator considering that it also consumes less code space and less stack RAM. If one of the undesirable properties of CCM must be avoided (not being online, lack of support for preprocessing of static AD), a good alternative is the EAX mode [2] which is described as a "cleaned-up CCM" by one of its authors and should have performance similar to CCM. The GCM mode, even though it has many good properties, does not appear to be adequate in software implementation for resource-constrained platforms since it requires very large lookup tables in order to offer performance comparable to other modes.

Some other relevant facts we have found are that Hummingbird-2 provided the fastest performance compared to the other modes using AES in software; that SGCM is 50% faster than GCM when using the AES accelerator and 25% when not; and that OCB3 and Hummingbird-2 in particular have a decryption performance remarkably slower than encryption (up to 50%).

Future work. It would be interesting to implement and compare lightweight encrypt-and-authenticate or authenticated encryption schemes such as LETTER-SOUP [17] and Rabbit-MAC [18] for the MSP430X. Another possible venue for research is to study the efficient implementation of authenticated encryption using the AES accelerator featured in other platforms such as the AVR XMEGA and n devices based on the ARM Cortex such as the EFM32 Gecko, STM32 and LPC1800.

References

1. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In: Advances in Cryptology — ASIACRYPT 2000, Lecture Notes in Computer Science, vol. 1976, pp. 531–545. Springer Berlin / Heidelberg (2000)
2. Bellare, M., Rogaway, P., Wagner, D.: The EAX mode of operation. In: Fast Software Encryption, Lecture Notes in Computer Science, vol. 3017, pp. 389–407. Springer Berlin / Heidelberg (2004)
3. Didla, S., Ault, A., Bagchi, S.: Optimizing AES for embedded devices and wireless sensor networks. In: Proceedings of the 4th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities. pp. 4:1–4:10 (2008)
4. Engels, D., Saarinen, M.J.O., Smith, E.M.: The Hummingbird-2 lightweight authenticated encryption algorithm. Cryptology ePrint Archive, Report 2011/126 (2011), `http://eprint.iacr.org/`
5. Gladman, B.: AES and combined encryption/authentication modes. `http://gladman.plushost.co.uk/oldsite/AES/` (2008)
6. Gouvêa, C.P.L., López, J.: Software implementation of pairing-based cryptography on sensor networks using the MSP430 microcontroller. In: Progress in Cryptology — INDOCRYPT 2009. Lecture Notes in Computer Science, vol. 5922, pp. 248–262. Springer Berlin / Heidelberg (2009)
7. Großschädl, J., Szekely, A., Tillich, S.: The energy cost of cryptographic key establishment in wireless sensor networks. In: Proceedings of the 2nd ACM symposium on information, computer and communications security. pp. 380–382. ASIACCS '07, ACM, New York, NY, USA (2007)
8. Jovanov, E., Milenkovic, A.: Body area networks for ubiquitous healthcare applications: Opportunities and challenges. Journal of Medical Systems pp. 1–10 (2011)

9. Krovetz, T., Rogaway, P.: The software performance of authenticated-encryption modes. In: Fast Software Encryption, Lecture Notes in Computer Science, vol. 6733, pp. 306–327. Springer Berlin / Heidelberg (2011)
10. Lim, S.Y., Pu, C.C., Lim, H.T., Lee, H.J.: Dragon-MAC: Securing wireless sensor networks with authenticated encryption. Cryptology ePrint Archive, Report 2007/204 (2007), `http://eprint.iacr.org/`
11. López, J., Dahab, R.: High-speed software multiplication in $\mathbb{F}_{2^m}$. In: Progress in Cryptology — INDOCRYPT 2000. Lecture Notes in Computer Science, vol. 1977, pp. 93–102. Springer Berlin / Heidelberg (2000)
12. McGrew, D., Viega, J.: The security and performance of the Galois/Counter Mode (GCM) of operation. In: Progress in Cryptology — INDOCRYPT 2004, Lecture Notes in Computer Science, vol. 3348, pp. 377–413. Springer Berlin / Heidelberg (2005)
13. Oliveira, L.B., Aranha, D.F., Gouvêa, C.P.L., Scott, M., Câmara, D.F., López, J., Dahab, R.: TinyPBC: Pairings for authenticated identity-based non-interactive key distribution in sensor networks. Computer Communications 34(3), 485–493 (2010)
14. Rogaway, P., Wagner, D.: A critique of CCM. Cryptology ePrint Archive, Report 2003/070 (2003), `http://eprint.iacr.org/`
15. Rogaway, P.: Method and apparatus for facilitating efficient authenticated encryption (2007), US patent 7200227
16. Saarinen, M.J.O.: SGCM: The Sophie Germain counter mode. Cryptology ePrint Archive, Report 2011/326 (2011), `http://eprint.iacr.org/`
17. Simplicio Jr, M.A., Barbuda, P.F.F.S., Barreto, P.S.L.M., Carvalho, T.C.M.B., Margi, C.B.: The MARVIN message authentication code and the LETTERSOUP authenticated encryption scheme. Security and Communication Networks 2(2), 165–180 (2009)
18. Tahir, R., Javed, M., Cheema, A.: Rabbit-MAC: Lightweight authenticated encryption in wireless sensor networks. In: Information and Automation, 2008. ICIA 2008. International Conference on. pp. 573–577 (2008)
19. Werner-Allen, G., Lorincz, K., Welsh, M., Marcillo, O., Johnson, J., Ruiz, M., Lees, J.: Deploying a wireless sensor network on an active volcano. IEEE Internet Computing 10, 18–25 (2006)
20. Whiting, D., Housley, R., Ferguson, N.: Counter with CBC-MAC (CCM) (2002), `http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html`