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 scheme that provides both confidentiality and authentication. In this work we describe an optimized implementation of authenticated encryption for the MSP430X family of microcontrollers. The CCM, GCM, SGCM, OCB3, Hummingbird-2 and MASHA authenticated encryption schemes were implemented at the 128-bit level of security and their performance was compared. The AES accelerator included 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 10 times of speedup. The CCM and OCB3 schemes were the fastest when using the AES accelerator while MASHA and Hummingbird-2 were the fastest when using only software.

Keywords: authenticated encryption, MSP430, AES, software implementation

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. 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 scheme. 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. There are many AE schemes; see e.g. [10] for a non-exhaustive list. Some AE schemes are built using a block cipher, in this case, they

* Supported by FAPESP, grant 2010/15340-3.

are also called AE modes. In this work, we follow the approach from [10] and compare the Counter with CBC-MAC (CCM) mode [18], the Galois/Counter Mode (GCM) [13] and the Offset Codebook (OCB3) mode [10]. We have also implemented the Sophie Germain Counter Mode [14], the Hummingbird-2 cipher [5] and the MASHA cipher [9]. 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 cipher (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 MASHA cipher is based on a stream cipher and claims to fill the gap for authenticated encryption algorithms based on stream ciphers which achieve a good balance between security and performance.

The goal of this work is to provide an efficient implementation and comparison of the aforementioned AE schemes (CCM, GCM, SGCM, OCB3, Hummingbird-2, and MASHA) 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; furthermore, some microcontrollers of this family feature an AES accelerator module which can encrypt and decrypt using 128-bit keys. Our main contributions are: (i) 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 schemes; (ii) to describe a high speed implementation of those AE schemes for the MSP430X, achieving performance 10 times faster for CCM using the AES accelerator instead of AES in software; (iii) to describe an efficient implementation of AES for 16-bit platforms; (iv) to show that CCM is the fastest of those schemes whenever a non-parallel AES accelerator is available; and (v) and to provide a comparison of the six AE schemes, with and without the AES accelerator. We remark that the results regarding the efficient usage of the AES accelerator can 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. 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 has distinct clock frequency, RAM and flash sizes.

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 therefore the cost of communicating with the accelerator is included in our timings.

3 Authenticated Encryption

An authenticated encryption scheme 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 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).

Many AE schemes 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 $D_K(B')$ where B' is also a block (usually from the ciphertext). The CCM, GCM, SGCM and OCB3 are based on block ciphers, while HB2 and MASHA are not.

It is possible to identify several properties of AE schemes; we offer a non-exhaustive list. The *number of block cipher calls* used in the scheme is an important metric related to performance. A scheme 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 schemes *only use the forward function* of the underlying block cipher (E_K), which reduces the size of software and hardware implementations. A scheme *supports preprocessing of static associated data (AD)* if the authentication of the AD depends only on the key and can be cached between

Table 1. Comparison of implemented AE schemes

Property	CCM	(S)GCM	OCB3	HB2	MASHA
Block cipher calls[a]	$2m + a + 2^b$	m	$m + a + 1^b$	—	—
... in key setup	0	1	1	—	—
Online	No	Yes	Yes	Yes[c]	Yes
Uses only E_K	Yes	Yes	No	—	—
Prepr. of static AD	No	Yes	Yes	No	N/A
Patent-free	Yes	Yes	No	No	No
Parallelizable	No	Yes	Yes	No	No
Standardized	Yes	(No) Yes	No	No	No
Input order	AD first	AD first	Any	AD last	N/A

[a] m, a are the number of message and AD blocks, respectively
[b] May have an additional block cipher call
[c] AD size must be fixed

different messages being sent (this is useful for a header that does not change). Some schemes are *covered by patents*, which usually discourages its use. A scheme is *parallelizable* if it is possible to process multiple blocks (or partially process them) in a parallel manner. Some schemes support *processing regular messages and AD in any order*, while some schemes require the processing of AD before the message, for example. The properties of the AE schemes implemented in this work are compared in Table 1.

Remarks about security. The weak key attack against GCM, pointed out by the author of SGCM [14], has probability $n/2^{128}$ of working, where n is the number of blocks in the message; this is negligible unless the message is large. There are related key attacks against Hummingbird-2 [2,19] which, while undesirable, can be hard to apply in practice since keys are (ideally) random. Finally, there is a key-recovery attack in the multi-user setting [3] that can be applied to all schemes in this paper; however, they can be avoided by using random nonces.

4 Efficient Implementation

We have written a fast software implementation of the AE schemes 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 5.30. For the AE modes based on block ciphers, we have used the AES with 128-bit keys both in software and using the AES accelerator. Our source code is available[1] to allow reproduction of our results.

The interface to the AES accelerator was written in assembly, along with a function to xor two blocks and another to increment a block.

[1] http://conradoplg.cryptoland.net/software/authenticated-encryption-for-the-msp430/

4.1 CCM

The CCM (Counter with CBC-MAC) mode [18] essentially combines the CTR mode of encryption with the CBC-MAC authentication scheme. For each message block, a counter is encrypted with the block cipher and the result xored to the message to produce the ciphertext; the counter is then incremented. The message is also xored to an "accumulator" which is then encrypted; this accumulator will become the authentication tag after all blocks are processed.

Its implementation was fairly straightforward, employing the assembly routines to xor blocks and increment the counter.

4.2 GCM

The GCM (Galois/Counter Mode) [13] employs the arithmetic of the finite field $\mathbb{F}_{2^{128}}$ for authentication and the CTR mode for encryption. For each message block, GCM encrypts the counter and xors the result into the message to produce the ciphertext; the counter is then incremented. The ciphertext is xored into an accumulator, which is then multiplied in the finite field by a key-dependent constant H. The accumulator is used to generated the authentication tag.

In order to speed up the GCM mode, polynomial multiplication was implemented in unrolled assembly with the López-Dahab (LD) [12] algorithm using 4-bit window and two lookup tables; it is described for reference in Appendix A, Algorithm 3. 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.

4.3 SGCM

The SGCM (Sophie Germain Counter Mode) [14] 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 $p = 2^{128} + 12451$.

Arithmetic in $\mathbb{F}_p$ can be carried out with known algorithms such as Comba multiplication. We follow the approach in [7] which takes advantage of the multiply-and-accumulate operation present in the hardware multiplier of the MSP430 family, also taking advantage of the 32-bit multiplier present in some MSP430X devices, including the CC430 series.

4.4 OCB3

The OCB3 (Offset Codebook) mode [10] 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). For each i-th message block, OCB3 computes the finite field multiplication of a nonce/key-dependent constant L_0 by the polynomial z^j, where j is the number of trailing zeros in the binary representation of the block index i; the result is xored into an accumulator Δ. This accumulator is xored to the message, encrypted, and the result is xored back with Δ to generate the ciphertext. The message block is xored into another accumulator Y, which is used to generate the tag.

A lookup table with 8 entries (128 bytes) was used to hold the some precomputed values of $L_0 \cdot z^j$. Two functions were implemented in assembly: multiplication by z (using left shifts) and the function used to compute the number of trailing zeros (using right shifts).

4.5 Hummingbird-2 (HB2)

The Hummingbird-2 [5] is an authenticated encryption algorithm which is not built upon a block cipher. It processes 16-bit blocks and was specially designed 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:

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

where S_1, S_2, S_3, S_4 are S-boxes and $\lll$ denotes the circular left shift of a 16-bit word. For each 16-bit message block, HB2 calls `WD16` four times, using as inputs different combinations of the message, state and key.

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 $f(x) = f_L[x \ \&\ \texttt{0xFF}] \oplus f_H[(x \ \&\ \texttt{0xFF00}) \gg 8]$. 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 $L^{-1}(x)$. In this case, we have used precomputed lookup tables L_L, L_H such that $L(x) = L_L[x \ \&\ \texttt{0xFF}] \oplus L_H[(x \ \&\ \texttt{0xFF00}) \gg 8]$. 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 $S^{-1}(x) = S_L^{-1}(x[0..7]) \mid (S_H^{-1}(x[8..15]) \ll 8)$.

4.6 MASHA

MASHA [9] is an authenticated encryption algorithm based on a stream cipher. Stream ciphers are interesting since they are often more efficient than block ciphers. However, many stream ciphers which also provide authentication either have security issues (e.g. Phelix) or performance issues. The MASHA authors propose the algorithm in order to attempt to fill this gap.

Our implementation was based on C source provided by the designers. We have changed it to reduce code size and memory footprint. The code stores the linear shift registers in circular buffers in order to avoid the actual shifts. The scheme requires multiplication, in $\mathbb{F}_{2^8}$, by four distinct constants. These are precomputed in a 256-element table which stores the multiplication of all bytes by these constants. Two such tables are required for each of the two distinct fields used by MASHA, totaling 2 KB. Since this is already large, we chose to use a byte-oriented approach for the `MixColumns` step instead of the 16-bit tailored code we will describe below. Therefore, the total space for the precomputed values becomes 2.75 KB.

4.7 Improving AES for 16-bit

We have used a software implementation of AES in order to perform comparisons with the hardware accelerator. Our implementation was based on the byte-oriented version from [6], but we have modified it to take advantage of the 16-bit platform. The first change was to improve the `AddRoundKey` function (which simply computes the xor of 128-bit blocks) in order to xor 16-bit words at a time. The second change was to improve the use of lookup tables as follows.

As it is well known, the input and output blocks of the AES can be viewed as 4×4 matrices in column-major order whose elements are in $\mathbb{F}_{2^8}$; and the AES function `SubBytes`, `ShiftRows` and `MixColumns` steps can be combined in a single one. In this step, the column j of the result matrix can be computed as

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = S[a_{0,j}] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus S[a_{1,j-1}] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus S[a_{2,j-2}] \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus S[a_{3,j-3}] \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix},$$

where e is the output matrix, a is the input matrix, S is the forward S-box, k is the round key matrix, and matrix indices are computed modulo four. Inspired by the 32-bit optimization of using four precomputed tables with 256 elements of with 4-byte each (totaling 4 KB), we employ the following tables:

$$T_0[a] = \begin{bmatrix} S[a] \cdot 02 \\ S[a] \end{bmatrix}, T_1[a] = \begin{bmatrix} S[a] \\ S[a] \cdot 03 \end{bmatrix}, T_2[a] = \begin{bmatrix} S[a] \cdot 03 \\ S[a] \cdot 02 \end{bmatrix}, T_3[a] = \begin{bmatrix} S[a] \\ S[a] \end{bmatrix}.$$

They consume 2 KB, half the size of the 32-bit version, providing a good compromise between the 8-bit and 32-bit oriented implementations. These tables allow

the computation of column e_j as

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \end{bmatrix} = T_0[a_{0,j}] \oplus T_2[a_{1,j-1}] \oplus T_1[a_{2,j-2}] \oplus T_3[a_{3,j-3}]\,,$$
$$\begin{bmatrix} e_{2,j} \\ e_{3,j} \end{bmatrix} = T_1[a_{0,j}] \oplus T_3[a_{1,j-1}] \oplus T_0[a_{2,j-2}] \oplus T_2[a_{3,j-3}]\,.$$

4.8 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 an efficient implementation using the module is to use this "delay slot" to carry out other operations that do not depend on the result of the encryption/decryption.

For example, in the CCM mode, the counter incrementation and the xor between the message and the accumulator can be carried out while the counter is being encrypted: the counter is written to the AES accelerator, the counter is incremented, we then wait for the result of the encryption and xor the result to the message when it is ready. In CCM it is also possible to generate the ciphertext (xor the encrypted result and the message) while the accumulator is being encrypted. In the GCM mode, it is possible to increment the counter while the counter is being encrypted. In the OCB3 mode, the xor between the message and the accumulator Y can be carried out while the message, xored to Δ, is being encrypted. For reference, these computations which can be carried out in the delay slot are marked in the algorithms of Appendix A.

5 Results

The performance of the implemented AE schemes was measured for the authenticated encryption and decryption-verification of messages with 16 bytes and 4 KB, along with the Internet Performance Index (IPI) [13], 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 and decryption-verification with 128-bit tags (except MASHA, which uses 256-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; this data can also be viewed as throughput in Figures 1 and 2, considering a 20 MHz clock. 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 schemes for different message lengths, in cycles per byte

	Using AES accelerator			Using AES in software		
Scheme	16 bytes	IPI	4 KB	16 bytes	IPI	4 KB
Encryption						
CTR[a]	26	23	23	195	194	193
CCM	116	38	36	778	381	375
GCM	426	183	180	696	320	314
SGCM	242	89	87	567	254	250
OCB3	144	39	38	469	209	205
HB2[b]				569	200	196
MASHA[b]				3 014	182	152
Decryption						
CTR[a]	26	23	23	195	194	193
CCM	129	47	46	781	380	375
GCM	429	183	180	699	319	314
SGCM	243	89	87	571	254	250
OCB3	217	48	46	510	245	242
HB2[b]				669	297	292
MASHA[b]				3 016	182	151

[a] Non-authenticated encryption scheme included for comparison

[b] 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 schemes; 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 schemes. Both CCM and OCB3 have almost the same speed, with CCM being around 4% faster. This is surprising, since that OCB3 essentially outperforms CCM in many platforms [10]. The result 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 180 cycles per byte to encrypt and 216 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

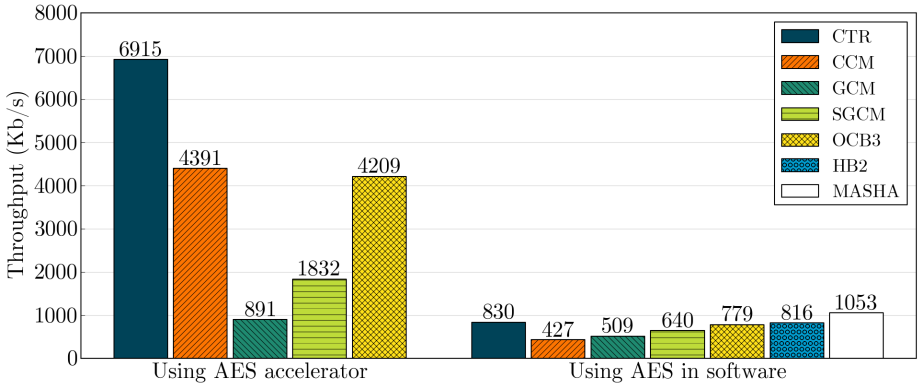

Fig. 1. Encryption throughput in Kbps of CTR and AE schemes for 4 KB messages at 20 MHz

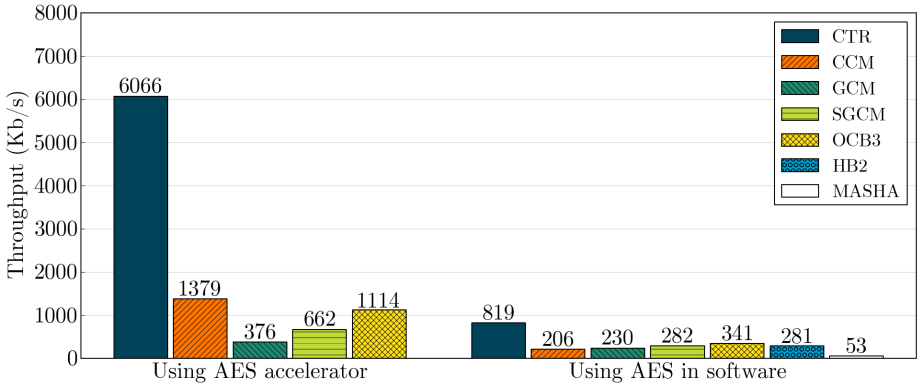

Fig. 2. Encryption throughput in Kbps of CTR and AE schemes for 16-byte messages at 20 MHz

SGCM is also faster than GCM, but the improvement is diluted to 20% with the software AES. The MASHA cipher is the fastest, followed by Hummingbird-2, which is 22% slower. Interestingly, Hummingbird-2 fails to outperform AES in CTR mode, which is surprising since it is specially tailored for the platform (of course, it must be considered that it provides authentication while AES-CTR by itself does not).

AES accelerator vs. AES in software. Using the AES accelerator, it is possible to encrypt in the CTR mode approximately 8 times faster than using AES in software; and it is possible to encrypt with CCM approximately 10 times faster for encryption and 8 times faster for decryption. The AES accelerator speedup for GCM, SGCM and OCB3 is smaller (around 1.7, 2.8, and 5.4, respectively), due to the larger software overhead.

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) as well as in MASHA. However, decryption is almost 18% slower for OCB3, since the underlying block cipher decryption is also slower than the encryption. 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.

Performance for small messages. The timings for 16-byte messages are usually dominated by the computation of nonce-dependent values. The CCM using software AES has the second worst performance 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, and CCM becomes the faster scheme. 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. Hummingbird-2 loses to OCB3 due to its larger nonce setup and tag generation. The greatest surprise is the MASHA performance which is almost four times slower than CCM, making it the slowest scheme for small messages. This result is explained by the fact that its nonce setup and tag generation are very expensive, requiring more than 20 state updates each (which take roughly the same time as encrypting ten 128-bit blocks).

Further analysis. In order to evaluate our AES software implementation, consider the timings from [4] (also based on [6]) 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 889 Kbps. We conclude that our 16-bit implementation is 24% faster then the byte-oriented implementation.

Table 3 lists the ROM and RAM usage for programs implementing AE schemes for both encryption and decryption, using the AES accelerator. The reported sizes refer only to the code related to the algorithms and excludes the benchmark code. 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 large due to the unrolled $\mathbb{F}_{2^{128}}$ multiplier, while 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 second 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. The MASHA cipher requires the largest code space, due to the many precomputed tables used; this can be reduced by sacrificing speed. When using the software AES implementation, 2 904 additional ROM bytes are required for CCM, GCM and SGCM (which use E_K only) and 5 860 additional ROM bytes are required for OCB3.

Table 3. ROM and RAM (stack) usage of AE schemes, in bytes. When using software AES, 2 904 additional ROM bytes are required for CCM, GCM and SGCM and 5 860 bytes for OCB3

	CTR	CCM	GCM	SGCM	OCB3	HB2	MASHA
ROM	130	1 094	4 680	2 172	1 724	3 674	5 602
RAM	100	258	886	322	538	196	499

5.1 Related work

A commercial 128-bit AES implementation for the MSP430 [8] achieves 340 cycles per byte for encryption and 550 cpb for decryption, in ECB mode, using 2536 bytes. Our implementation provides 180 cpb and 216 cpb, respectively, but uses 5860 bytes. With space-time tradeoffs, it should be feasible to achieve similar results, but we have not explored them.

Simplicio Jr. et al. [16] have implemented EAX, GCM, LetterSoup, OCB2 and CCFB+H for the MSP430, using Curupira as the underlying block cipher. The EAX mode is [1] is described as an "cleaned-up" CCM and has similar performance. The authors report the results in milliseconds, but do not state the clock used. Assuming a 8 MHz clock, their timings (in cycles per byte, considering their timings for 60-byte messages and our timings for 16-byte messages) are 1 733 cpb for EAX, 5 133 cpb for GCM, 1 680 cpb for LetterSoup, 1 506 cpb for OCB2 and 2 266 cpb for CCFB+H with 8-byte tag. Our CCM is 2.2 times faster than their EAX, while our GCM is 7.3 times faster, and our OCB3 3.2 times faster than their OCB2. This difference can probably be explained by the fact that the authors have not optimized the algorithms for performance.

In [4], 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 [11] 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; while our HB2 is 9.2 times slower and 5.1 times smaller.

The Hummingbird-2 timings reported for the MSP430 in its paper [5] are about 6% and 2% faster for encryption and decryption than the timings we have obtained. However, the 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. Furthermore, by completely unrolling the encryption and decryption functions, we were able to achieve timings 3% and 4% faster than theirs, increasing code size by 296 and 432 bytes, respectively.

6 Conclusion and Future Work

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 scheme 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 analogue AES module.

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 [1] and should have performance similar to CCM. Considering software-only schemes, it is harder to give a clear recommendation: SGCM, OCB3 and HB2 provide good results, with distinct advantages and downsides. 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 be competitive.

Some other relevant facts we have found are that Hummingbird-2 is slower than AES; that SGCM is 50% faster than GCM when using the AES accelerator and 20% when not; and that OCB3 and Hummingbird-2 in particular have a decryption performance remarkably slower than encryption (18% and 50% respectively). MASHA has great speed for large enough messages (29% faster than the second fastest, HB2) but very low performance for small messages (almost 4 times slower than the second slowest, CCM). For this reason, we believe there is still the need for a fast, secure and lightweight authenticated encryption scheme based on a stream cipher.

For future works it would be interesting to implement and compare lightweight encrypt-and-authenticate or authenticated encryption schemes such as LetterSoup [15] and Rabbit-MAC [17] 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 devices based on the ARM Cortex such as the EFM32 Gecko, STM32 and LPC1800.

References

1. 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)
2. Chai, Q., Gong, G.: A cryptanalysis of HummingBird-2: The differential sequence analysis. Cryptology ePrint Archive, Report 2012/233 (2012), http://eprint.iacr.org/
3. Chatterjee, S., Menezes, A., Sarkar, P.: Another look at tightness. In: Selected Areas in Cryptography, Lecture Notes in Computer Science, vol. 7118, pp. 293–319. Springer Berlin / Heidelberg (2012)
4. 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)
5. Engels, D., Saarinen, M.J.O., Smith, E.M.: The Hummingbird-2 lightweight authenticated encryption algorithm. In: RFID Security and Privacy, Lecture Notes in Computer Science, vol. 7055. Springer Berlin / Heidelberg (2011)
6. Gladman, B.: AES and combined encryption/authentication modes. http://gladman.plushost.co.uk/oldsite/AES/ (2008)
7. Gouvêa, C.P.L., López, J.: Efficient software implementation of public-key cryptography on sensor networks using the MSP430X microcontroller. Journal of Cryptographic Engineering 2(1), 19–29 (2012)
8. Institute for Applied Information Processing and Communication: Crypto software for microcontrollers - Texas Instruments MSP430 microcontrollers. http://jce.iaik.tugraz.at/sic/Products/Crypto_Software_for_Microcontrollers/Texas_Instruments_MSP430_Microcontrollers (2012)
9. Kiyomoto, S., Henricksen, M., Yap, W.S., Nakano, Y., Fukushima, K.: Masha — low cost authentication with a new stream cipher. In: Information Security, Lecture Notes in Computer Science, vol. 7001, pp. 63–78. Springer Berlin / Heidelberg (2011)
10. 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)
11. 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/
12. 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)
13. 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)
14. Saarinen, M.J.O.: SGCM: The Sophie Germain counter mode. Cryptology ePrint Archive, Report 2011/326 (2011), http://eprint.iacr.org/
15. 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)

16. Simplicio Jr., M.A., de Oliveira, B.T., Barreto, P.S.L.M., Margi, C.B., Carvalho, T.C.M.B., Naslund, M.: Comparison of authenticated-encryption schemes in wireless sensor networks. In: 2011 IEEE 36th Conference on Local Computer Networks (LCN). pp. 450–457 (2011)
17. 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)
18. Whiting, D., Housley, R., Ferguson, N.: Counter with CBC-MAC (CCM) (2002), http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html
19. Zhang, K., Ding, L., Guan, J.: Cryptanalysis of Hummingbird-2. Cryptology ePrint Archive, Report 2012/207 (2012), http://eprint.iacr.org/

A Algorithms

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.

Algorithm 1 CCM encryption

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

$B_0, A_1, \ldots, A_a, M_1, \ldots, M_m \leftarrow \texttt{format}(N, A, M)$
$Y \leftarrow E_K(B_0)$
for $i \leftarrow 1$ **to** a **do**
  $Y \leftarrow E_K(A_i \oplus Y)$
end for
$J \leftarrow \texttt{init\_ctr}(N)$
$S_0 \leftarrow E_K(J)$
$J \leftarrow \texttt{inc}(J)$
for $i \leftarrow 1$ **to** m **do**
  $U \leftarrow E_K(J)$
  $J \leftarrow \texttt{inc}(J)$ {delay slot}
  $S \leftarrow M_i \oplus Y$ {delay slot}
  $Y \leftarrow E_K(S)$
  $C_i \leftarrow M_i \oplus U$ {delay slot}
end for
$T \leftarrow Y[0..t-1] \oplus S_0[0..t-1]$

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). The LD multiplication with two tables, used in the field multiplication, is described in Algorithm 3.

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: $\quad Y \leftarrow (A_i \oplus Y) \cdot H$
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: $\quad U \leftarrow E_K(J)$
13: $\quad J \leftarrow \texttt{inc}(J)$ {delay slot}
14: $\quad C_i \leftarrow M_i \oplus U$
15: $\quad Y \leftarrow (C_i \oplus Y) \cdot H$
16: **end for**
17: $L \leftarrow [len(A)]_{64} \,||\, [len(M)]_{64}$
18: $S \leftarrow (L \oplus Y) \cdot H$
19: $T \leftarrow (S \oplus S_0)[0..t-1]$

OCB3 is described in Algorithm 4, where the function `init_delta` derives a value from the nonce and it may require a block cipher call, as explained later. The function $\texttt{ntz}(i)$ returns the number of trailing zeros in the binary representation of i (e.g. $\texttt{ntz}(1) = 0$, $\texttt{ntz}(2) = 1$). The function $\texttt{getL}(L_0, x)$ 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.

Algorithm 3 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

Algorithm 4 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: $\Delta \leftarrow \texttt{init\_delta}(N, K)$
8: **for** $i \leftarrow 1$ **to** m **do**
9: $\Delta \leftarrow \Delta \oplus \texttt{getL}(L_0, \texttt{ntz}(i))$
10: $U \leftarrow E_K(M_i \oplus \Delta)$
11: $Y \leftarrow Y \oplus M_i$ {delay slot}
12: $C_i \leftarrow U \oplus \Delta$
13: **end for**
14: $\Delta \leftarrow \Delta \oplus L_\$$
15: $F \leftarrow E_K(Y \oplus \Delta)$
16: $G \leftarrow \texttt{hash}(K, A)$
17: $T \leftarrow (F \oplus G)[0..t-1]$