Optimizing AES for Embedded Devices and Wireless Sensor Networks

Shammi Didla
sdidla@purdue.edu

Aaron Ault
ault@purdue.edu

Saurabh Bagchi
sbagchi@purdue.edu

Center for Wireless Systems and Applications (CWSA)
Purdue University, West Lafayette, IN 47906, USA

ABSTRACT

The increased need for security in embedded applications in recent years has prompted efforts to develop encryption algorithms capable of running on resource constrained systems. The inclusion of the Advanced Encryption Standard (AES) in the IEEE 802.15.4 Zigbee protocol has driven its widespread use in current embedded platforms. We propose an implementation of AES in a high-level language (C in this case) that is the first software-based solution for 16-bit microcontrollers capable of matching the communication rate of 250 kbps specified by the Zigbee protocol, while also minimizing RAM and ROM usage. We discuss a series of optimizations and their effects that lead to our final implementation achieving an encryption speed of 286 kbps, RAM usage of 260 bytes, and code size of 5160 bytes on the Texas Instruments MSP430 microprocessor. We also develop rigorous benchmark experiments to compare other AES implementations on a common platform, and show that our implementation outperforms the best available implementation by 85%.

Categories and Subject Descriptors

E.3 [**Data Encryption**]: Standards—*AES optimization, embedded devices*

General Terms

Algorithms, Security, Performance

Keywords

AES, encryption, embedded optimizations, secure sensor networks, CC2420, MSP430, Zigbee security

1. INTRODUCTION

The proliferation of wireless sensor networks (WSN) in recent years has prompted increased interest in secure communications for embedded devices. Wireless sensor nodes are inherently resource-constrained in terms of processor speed, bandwidth, energy usage, code space, and RAM size. Therefore, there is a need for secure encryption/decryption implementations that have a small footprint while performing at speeds comparable to the radio transmission bitrate on a low-speed processor.

The Advanced Encryption Standard (AES) became the standard for encryption to protect sensitive information by all U.S. government organizations on May 26, 2002 [4]. Its inclusion in the IEEE 802.15.4 [5] standard as the standard encryption protocol for ZigBee makes AES ideal for use in WSNs.

According to the IEEE 802.15.4 specification, Low Rate - Wireless Personal Area Networks (LR-WPAN) have a maximum over-the-air data rate of 250 kbps. Therefore, it is important for the encryption to match this rate to achieve optimal wireless communication. After carefully reviewing and experimenting with previous works, we came to the conclusion that no previous software scheme is able to encrypt data using AES at a rate of 250 kbps or higher. Moreover, there was considerable disagreement among various research groups about the performance and memory footprint of AES implementations. The memory footprint consists of RAM usage and ROM usage. RAM is often a highly constrained resource on the embedded platforms (e.g., the Crossbow Mica2 mote has 4 KB and the MSP430 chip has up to 10 KB). The ROM memory is used to hold the program and therefore it is desirable to limit its usage by the cryptographic functions.

In this paper we show that AES can indeed be rate matched with the radio communication speed, thus making it practical for use in WSNs. Our fastest implementation of AES achieved an encryption speed of 286 kbps and required 5160 bytes of ROM and 260 bytes of RAM. To the best of our knowledge, no previous implementation on a similar platform has been able to match this rate. At this encryption speed, it is also possible to eliminate latency due to the encryption process in a IEEE 802.15.4-compliant WSN and therefore use AES on a continuous stream of 128-bit data blocks.

To achieve this speed, we applied various optimization techniques to Gladman's AES implementation [9] for low resource platforms. We evaluated the effects of specializing the code (SPECIAL) for AES-128 by removing code that accommodates for variable key length, varying the data type (DATASZ) that holds the state and key, eliminating function calls by integrating all functional blocks into a single function (INLINE), unrolling looping constructs by taking

TRIDENTCOM 2008, 17th – 20th Mar 2008, Innsbruck, Austria.
 ISBN 978-963-9799-24-0
DOI 10.4108/icst.tridentcom.2008.10409

a copy-paste approach for repeated operations (UNROLL), reducing moving data around memory by restructuring the original implementation (REDMEM), eliminating the use of local buffers to hold the state (LOCBUF), using a global variable to hold the key schedule (GLOB), generating the key for each round during the encryption process instead of precomputing it and storing it in RAM (OTFK) and using 16-bit memory writes in the `MixColumns` transformation (MIX16). We also examined how these optimizations interact and occasionally conflict with compiler optimizations and their net effect on performance, ROM and RAM. Based on our analysis, we recommend the use of SPECIAL, DATASZ (64-bits), INLINE, LOCBUF, REDMEM and GLOB for best performance and additionally apply OTFK to optimize for RAM usage.

To understand how the flexibility and cost-effectiveness of our software implementation written in C compares to the performance advantage of using a hardware implementation, we tested both solutions in a real-time wireless communication scenario using evaluation boards equipped with Zigbee-compliant transceiver chips. We also studied and evaluated three past attempts at optimizing AES in software that represent state-of-the-art in optimized encryption implementations for an embedded platform. By using a common platform, a high precision oscilloscope to accurately measure time to within $\pm 5\mu s$ and by rigorously standardizing tests across different implementations, we were able to reliably and quantitatively compare different implementations and evaluate their performance.

We developed and tested all our code on a Texas Instruments' MSP430 microcontroller unit (MCU) running at 8 MHz. The MSP430 family of microcontrollers is a popular choice for several sensor nodes such as the Eyes Node [1] and the T-Mote Sky [2]. It has a 16-bit RISC Core with a flexible clock system and its low power consumption of about 250 μA/MIPS-active make it ideal for WSNs. We also bought evaluation boards that interfaced the MSP430 to a Chipcon CC2420 transceiver chip. The CC2420 is IEEE 802.15.4 Zigbee-compliant, has an effective data rate of 250 kbps and has support for hardware MAC encryption (AES-128). We used this setup to evaluate the performance of a hardware implementation of AES and also compare it to our software solution.

Our main contributions in this paper are showing that AES can perform at the radio communication rate by implementing the fastest software solution to date, presenting an in-depth analysis of various optimization techniques required to achieve this speed and performing a rigorous and quantitative benchmark of multiple software and hardware solutions.

The rest of the paper is organized as follows. In the next section, we give details about the other optimized AES implementations that we chose to evaluate. In Section 3, we give a brief description of AES, discuss the key aspects of Gladman's implementation and the IEEE 802.15.4 MAC sublayer security specification. In Section 4, we describe all the computational and memory optimizations we used, discuss the intuition behind and predict the effect of each. In Section 5, we describe our experimental setup and methodology. In Section 6 we present the results of our experimentation and in the last section, we conclude this paper and discuss future work.

2. RELATED WORK

We identified three other fast implementations of AES and obtained the source code from their authors. In addition to these three, we also chose to evaluate the implementation provided in the freely available Zigbee Stack for the CC2420. Below is a brief description, focus and published performance figures of each of these implementations.

In [12], the authors have benchmarked various block Ciphers including Rijndael (AES) on a 16-bit MSP430 microcontroller. Their implementation is based on code from the open source OpenSSL library. It is heavily modified and compiled with the commercial IAR Workbench compiler. They have speed-optimized and size-optimized versions of each implementation running in Cipher-Block Chaining Mode (CBC), Cipher Feedback Mode (CFB), Output Feedback Mode (OFB) and Counter Mode (CTR). Their estimate shows that AES performs best in OFB mode taking 3127 clock cycles to encrypt a 128-bit block of plaintext while taking up 12860 bytes of code memory (ROM) and 70 bytes of data memory (RAM). Their size-optimized AES implementation takes 4231 clock cycles to encrypt a 128-bit block of plaintext taking up 12616 bytes of ROM and 70 bytes of RAM.

In [14], the authors focus on the need for a compact implementation. Their implementation requires 3322 bytes in ROM and 177 bytes in RAM. However, to achieve low code size they have sacrificed performance. Their implementation takes 3.75 ms to encrypt a 128-bit block of plaintext on a 16-Bit MSP430 microcontroller running at 4 MHz.

In [6], the authors implement AES on a sensor node based on the 8-bit Atmel ATmega 128L microcontroller running at 8 MHz. They have based their implementation on Gladman's code that was cited in the AES proposal. Their implementation can encrypt a 128-bit block of plaintext in 0.857 ms.

Texas Instruments has made available a Zigbee Stack for the boards using the MSP430 with the CC2420. Even though the CC2420 has hardware support for AES, the stack includes a software implementation of the AES-128 encryption algorithm. We chose to add this to our evaluation list because we expect wide use of this implementation by WSN developers. No performance figures are provided with the implementation.

Figure 7 and Table 5 in Section 6 summarize the published results of each of these implementations while also comparing them to the results obtained using our own testing methods.

3. BACKGROUND

3.1 Advanced Encryption Standard

Rijndael Cipher, developed by Joan Daemen and Vincent Rijmen was accepted as the Advanced Encryption Standard on November 26, 2001. It is a symmetric-key block cipher with a block length of 128-bits and a flexible key length of 128, 192 or 256 bits. This section gives an overview of how AES works.

3.1.1 Encryption/Decryption Algorithm

A series of permutations and substitutions are applied to the plaintext for encryption. Fig. 1 illustrates the overall structure of the algorithm [13]. There are 4 main transformations used in this process. Each transformation is applied

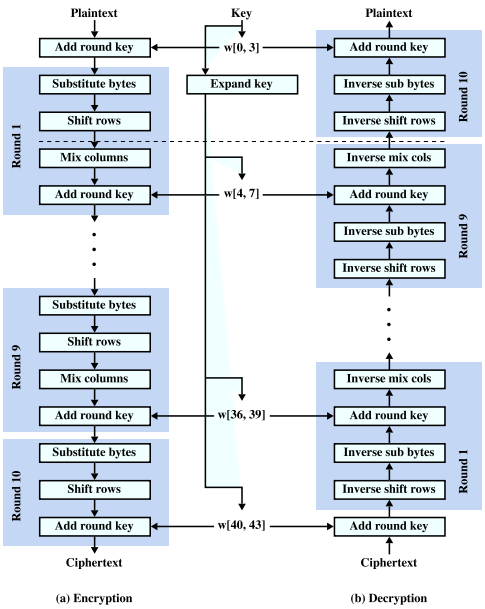

Figure 1: Overall Structure of AES

to a 4×4 byte matrix called the *State*. These transformations are described below:

- `SubBytes:` Each byte in the state is substituted by a byte from a 256-byte look-up table called the s-box.
- `ShiftRows:` The bytes in each of the 4 rows in the state are rotated by $(n-1)$ where n represents the row number from 1 to 4.
- `MixColumns:` The state can be considered to be a 4×4 matrix and this transformation can be achieved by multiplying this matrix by:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

  This multiplication is done in $GF\left(2^8\right)$[1]
- `AddRoundKey:` In this transformation, the *round key* is simply added to the state. In $GF\left(2^8\right)$, adding is equivalent to a bitwise exclusive-or operation.

The encryption process consists of initially applying `AddRoundKey` and 10, 12 or 14 *rounds* depending on the length of the key. Each round except the last one consists of applying the 4 transformations to the state. In the last round, only the `SubBytes`, `ShiftRows` and `AddRoundKey` transformations are applied.

[1]All arithmetic in Rijndael is done in a Galois Field with 256 Elements

Procedure	Times Called
`KeyExpand`	1
`SubBytes`	9
`ShiftRows`	9
`MixColumns`	8
`AddRoundKey`	10

Table 1: Frequency of transformations in applying AES-128 to a single data block

3.1.2 *The Key Expansion*

The cipher key is expanded to generate a different key for each round. Similar to the State, the key is also considered to be a two-dimensional matrix consisting of 4 rows. Each column is considered to be a 4-byte word. The expansion is achieved by applying `SubWord` and `RotWord` transformations and addition in $GF\left(2^8\right)$ of `RCon[]`, a constant word array. These operations are described below:

- `SubWord:` Similar to the `SubBytes` transformation, this is done by substituting each byte in the word with a byte from a 256-byte substitution box.
- `RotWord:` This transformation cyclically shifts the bytes of a word one place upwards.

Since the key expansion differs slightly for 128-, 192- and 256-bit keys, and our implementation in this paper deals with only 128-bit keys, we will discuss only 128-bit key expansion here. We chose to limit ourselves to AES-128 because we think it provides sufficient data protection for Wireless Sensor Networks. Moreover, the hardware module in Chipcon CC2420 is also limited to AES-128.

For AES-128, the expanded key consists of 176-bytes (44 words). The first 4 words of the expanded key consist of the original cipher key. Every word after that is equal to the sum of the previous word and the word 4 positions earlier. For words in positions that are multiples of 4, the `SubWord` and `RotWord` transformations are applied before applying the above described exclusive-or. After the exclusive-or, another exclusive-or with the `RCon[]` associated with the round is applied.

3.1.3 *Profiling*

Table 1 is a frequency distribution table of the different transformations in the encryption process. This serves as a good starting point in the analysis of the algorithm for optimization.

3.2 Brian Gladman's Low Resource Implementation

In this section, we discuss some of the important aspects of Gladman's implementation

3.2.1 *Use of Look-Up Tables*

As mentioned in the AES proposal, all modular mathematics of AES can be reduced to a series of table look-ups and exclusive-or operations. Almost all implementations that we looked at including Gladman's take this approach. Therefore, we concluded that using look-up tables was the only viable option in any practical scenario.

Level	Attribute	Confidentiality	Description
0x00	None	NO	No security
0x01	MIC-32	NO	Auth (CBC-MAC) 32 bit MIC
0x02	MIC-64	NO	Auth (CBC-MAC) 64 bit MIC
0x03	MIC-128	NO	Auth (CBC-MAC) 128 bit MIC
0x04	ENC	YES	Enc (Counter mode AES)
0x05	ENC-MIC-32	YES	Enc + Auth (CCM-Mode) 32 bit MIC
0x06	ENC-MIC-64	YES	Enc + Auth (CCM-Mode) 64 bit MIC
0x07	ENC-MIC-128	YES	Enc + Auth (CCM-Mode) 128 bit MIC

Table 2: Security modes specified in IEEE 802.15.4

Gladman's implementation had three 256-byte look-up tables used for encryption and five 256-byte look-up tables for decryption.

3.2.2 Combination of Transformations

Gladman combined the `MixColumns` and `SubBytes` transformations as well as the `ShiftRows` and `SubBytes` transformations into two functions. These combinations are possible because the shifting of rows and mixing of columns are always the same and are independent of the contents of the state. A large number of memory moves are eliminated by combining these transformations with the `SubBytes` transformation.

This technique was developed by Mark Malbrain and his contribution is acknowledged in Gladman's code.

3.2.3 Tuning Options

Gladman's code has 3 options which can be changed prior to compiling the code. These options are made possible using conditional preprocessor directives and modify the code considerably before compilation. These options can be activated/deactivated by using the #define preprocessor directive. These are briefly described below:

- `HAVE_MEMCPY`: Defining this directs the compiler to take advantage of the `memcpy` function in the compiler's standard library
- `HAVE_UINT32`: Defining this directs the compiler to take advantage of 32-bit data types if available on the target platform
- `VERSION_1`: Defining this makes extensive use of local buffers within functions instead of accessing data through pointers

3.3 IEEE 802.15.4 Security Specification

The IEEE 802.15.4 standard was first released in 2003 and revised in 2006. It includes Wireless Medium Access Control (MAC) as well as Physical layer specifications. Security is specified as part of the MAC sublayer. Since most WSNs fall within the category of LR-WPANs, compliance with this standard ensures reliability, compatibility and scalability of the network. There are a total of 8 security modes of which 4 ensure data confidentiality. These modes are listed in Table 2.

All four modes that ensure data confidentiality use AES as the underlying block cipher function. Level 0x04 uses AES in counter mode whereas levels 0x05 through 0x07 use AES in CCM mode. Moreover, CBC-MAC is a cipher based authentication scheme that in this case, once again, uses AES as the block cipher. For more information on AES modes of operation, refer to [7].

Therefore all security modes (except 0x00) rely on AES as the block cipher with a block length of 128-bits. Irrespective of the mode, a fast implementation of the AES block cipher is as essential building block of any secure IEEE 802.15.4 compliant system.

4. OPTIMIZATIONS

Gladman's code implements AES for key sizes of 128 and 256 bits. The code also includes on-the-fly key generation option. We use his code without the use of any tuning options as our baseline implementation. In this section, we discuss in detail the optimizations we applied and the intuition behind each optimization.

Since the integrity of the AES algorithm is of prime importance, these optimizations only aim at streamlining the program flow so as to achieve the same mathematical operation using fewer processor instructions. This ensures that our optimized implementation is in strict accordance with the AES specification. We verify the correctness of each implementation by comparing them to the test values included in [4].

4.1 Manual Optimizations

4.1.1 Specialization of Code (SPECIAL)

As mentioned before, the baseline implementation is a generic implementation capable doing AES-128 as well as AES-256. This definitely adds to the code size and hurts the performance of the key expansion process. By focusing on AES-128 and making the code less generic, we can eliminate a lot of conditional constructs and thereby substantially decrease the code size and improve performance.

4.1.2 Varying Data Type Size (DATASZ)

One of Gladman's tuning options is to take advantage of 32-bit data types (if available) instead of the 8-bit data types. Our compiler's largest data type is 64-bits. We expect the use of 8-bit data types to be highly inefficient since we are operating on a 16-bit platform. The use of data types larger than 16-bits is tested though we do not expect a substantial performance gain beyond 16-bit types. We expect this to show a substantial effect in the `AddRoundKey` transformation since the processor can exclusive-or 16-bits at a time instead of doing 8-bits at a time. Our profiling data (Table 1) also shows that `AddRoundKey` is the most frequently used transformation.

4.1.3 Function-Inlining (INLINE)

Function inlining is a very common optimization technique. Instead of organizing code into discrete functions which can be reused as and when required, all the functional blocks of the algorithm are coded into a single function. This eliminates the need to save the state of the function onto the stack and subsequently retrieve it. Function inlining improves performance but also increases code size if there is repeated use of code segments that perform the same set of operations on different data sets. In AES, each transformation is called only once per round. Therefore, when using a loop construct, we expect to see only a slight increase in code size. However, if the loops are unrolled and the functions are inlined, the code size might increase substantially. To have more control over function inlining, we chose to manually do this as opposed to using the compiler option.

4.1.4 Loop Unrolling (UNROLL)

Loop unrolling is another very commonly used optimization technique which has similar effects as function inlining. Instead of using a looping construct to iterate multiple times and use an index to perform the same operation on different sets of data, the code to perform the operation is copy-pasted multiple times. Since AES-128 has 10 rounds, 9 of which consist of exactly the same set of transformations, any performance gain from loop unrolling will be 9 fold. By manually unrolling the loops, we can also eliminate calculating array indices based on the loop counter. This usually results in better performing code while adding substantially to the code size.

4.1.5 Reducing Memory Moves (REDMEM)

The baseline implementation has several `memcpy` function calls while operating on the input state. We can eliminate copying of data from one memory location to the other by restructuring the code so that each transformation function saves its output in a location that is used as the input for the next transformation. To be able to do this, we will need to perform two sets of transformations on the state during each iteration of the main loop. This approach is illustrated in Fig. 2.

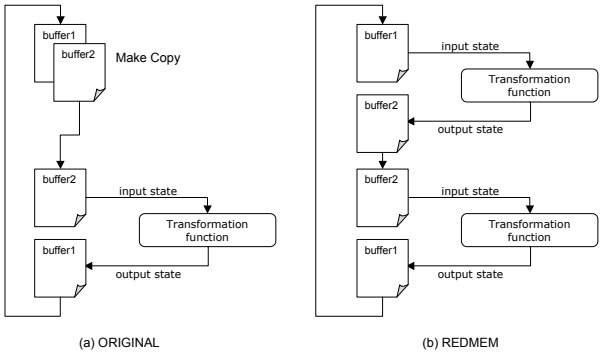

Figure 2: Restructuring the program to reduce data movement (REDMEM)

4.1.6 Eliminate Local Function Buffers (LOCBUF)

Gladman's code had a tuning option to either copy the state into a local buffer and then operate on it or access the state by passing a pointer to it to the transformation function. Gladman suggested that the performance implication of having a local buffer would depend on the platform. This is due to the fact that different microcontrollers have different memory addressing modes. Since the MSP430 family has a wide range of addressing capabilities, we expect passing of pointers to be more efficient since we eliminate copying the data.

4.1.7 Use of Global Variables (GLOB)

Due to the above mentioned issue with addressing modes, global variables have an advantage over local variables since their address can be precomputed before runtime by the compiler. AES is very efficient in terms of memory usage.

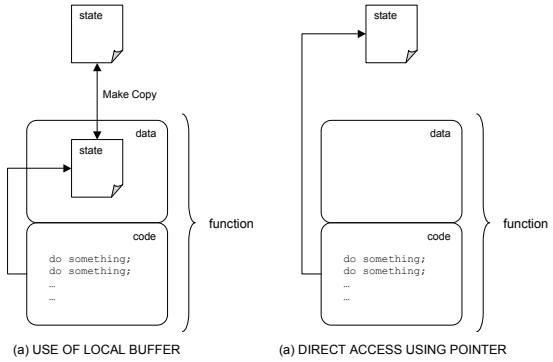

Figure 3: Alternative approach to accessing the state from a function

It uses a minimal amount of memory for all its transformations and can operate with the help of a 128-bit extra buffer to store the temporary state. The only scope for use of a global variable is to store the 176-byte key schedule since this is accessed by multiple functions at all stages of the encryption/decryption process.

4.1.8 On-the-fly-key Generation (OTFK)

For encryption/decryption, a 16-byte key gets expanded into a 176-byte key which can then be reused for 128-bit blocks of plaintext data. The 176-byte array that holds the key accounts for a very large percentage of the RAM requirement of AES. In cases where data memory is more important that performance, the need for a 176-byte array can be eliminated by generating the key on the fly during each round. This technique becomes proportionally less efficient compared to the *pre-keyed* version with increasing size of the plaintext to be encrypted with the same key.

4.1.9 16-bit Memory Writes in MixColumns (MIX16)

As described in Section 3, the `MixColumns` transformation can be implemented using XOR operations and table lookups. Each 8-bit entry in the state is replaced by XORing four 8-bit values from precomputed tables. To reduce the number of memory writes, we can compute a 16-bit entry for the state by using a 8-bit shift and an OR operation on two sets of four 8-bit values. The effectiveness of this optimization depends on the speed of memory writes versus the cost of bitwise-or and bitwise-shift operations. For example, if A_8, B_8 are 8-bit numbers and $v_8[\,]$ and $v_{16}[\,]$ are arrays of 8-bit and 16-bit elements respectively, the statements:

$$v_8[0] = A_8;$$
$$v_8[1] = B_8;$$

have the same effect as

$$v_{16}[0] = A_8|(B_8 \ll 8);$$

4.2 Compiler Optimizations

The msp-gcc compiler we are using is based on version 3.2.3 of GNU GCC. It includes several compiler optimizations which are broadly divided into 4 categories:

- O1 (level 1): compiler tried to reduce code size and execution time

- O2 (level 2): compiler turns on all optimizations except loop unrolling, function inlining and register renaming
- O3 (level 3): compiler turns on all optimizations including loop unrolling, function inlining and register renaming
- Os (Optimize for size): compiler turns on all O2 optimizations that do not increase code size

Since the primary focus of our paper is to optimize for speed, we compiled all versions of our implementation with the O3 option. Since O3 turns on a large number of compiler optimizations, in some cases, it cancels out the effect of our manual optimizations. To understand and analyze such cases better, we also tested all our code without the use of any compiler optimizations. This method enabled us to get a clear idea of the effects of O3 as well as our manual optimizations. We discuss each optimization and its effect in Section 6.

5. EXPERIMENTAL METHOD

5.1 The Setup

As mentioned before, we chose to develop on the MSP430 platform. These microcontrollers are available in different configurations. Key features of the MSP430F1611 MCU we used are listed below:

- Clock frequency of 8 MHz
- 48 KB ISP Flash ROM
- 10 KB RAM
- Power consumption: $330\mu A$ at 1 MHz, 2.2 V

Our main reason for choosing the MSP430F1611 is the commercial availability of an evaluation board from SoftBaugh Inc. The evaluation board interfaces the MSP430 to a Chipcon CC2420 transceiver chip as specified in technical documentation provided by TI [11]. Key features of the CC2420 are listed below:

- 2.4 GHz IEEE 802.15.4 compliant RF transceiver
- 250 kbps effective data rate
- Low power consumption: 17.4 - 19.7 mA, 2.1 - 3.6 V
- 4-wire SPI interface
- Serial clock up to 10 MHz
- Hardware MAC encryption (AES-128)

All code was tested on the Softbaugh DZ1611 Zigbee demo boards (Fig. 4, taken from [3]). The MSP430 can be programmed by either a JTAG interface provided on the board or a custom BootStrap Loader (BSL) interface.

Some changes were made to the board:

- The default 6 MHz crystal oscillator was replaced with a 8 MHz crystal to get peak performance from the microcontroller.

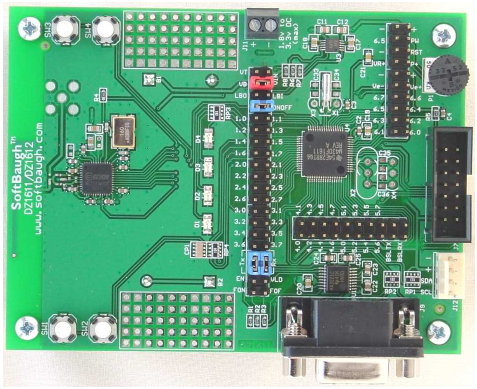

Figure 4: SoftBaugh DZ1611 Zigbee demo board

- Four pull-up resistors were added to the 4-wire SPI interface between the CC2420 and the MSP430. This was done because the CC2420 operates on *active low* signals.

We chose to develop using the open source mspgcc toolchain. This is a port of the gcc compiler and a subset of GNU tools to the MSP430 platform. We used mspgcc version 3.2.3.

5.2 Metrics

Upon reviewing previous works, we see significant disagreement with regard to the resource requirements of AES as well as its performance capability. Several reasons can account for this:

1. Measurement methodology
2. Differences in implementation
3. Hardware platform
4. Software tools

In this section, we discuss an accurate method for measuring parameters that are of interest to us: namely, RAM, ROM and Speed. This will help us better evaluate AES and its implementations.

5.2.1 ROM

or Code memory is the flash memory used by the program when it is loaded onto the MCU. This is the most straightforward parameter to measure. When the C code is compiled, the compiler generates several segments. The TEXT segment contains executable instructions and global constants and is loaded into the the MCU's ROM. The size of each segment and its target physical address can be obtained using the msp430-objdump utility. We are interested in measuring the code memory used by the AES implementation only and not the whole program which includes the `main()` function with the code to initialize the MCU and call the encryption/decryption functions. To do this we compile the test code without the encryption/decryption code to obtain the size contributed by the main function and subtract this from the size of the TEXT segment of the original test code. This value is the best estimate of the ROM requirement of the AES module.

5.2.2 RAM

or Data memory refers to the volatile, high-speed onboard memory of the MCU. This resource is extremely limited on most embedded systems. It is also hard to measure because in addition to global variables, we also need to take into account the variable stack size. Accurately measuring the stack size has been a difficult challenge for embedded systems developers. Yet it is important because bugs resulting from stack overflows are unpredictable and hard to find. We chose to use a very accurate and reliable method that tends to be somewhat involved. The steps required to calculate the stack size are given below:

1. Compile the code with the -g option to include debugging information for use by msp430-gdb (MSP430 port of the GNU debugger)
2. Load the code on the MCU through msp430-gdb using the JTAG interface for real-time debugging
3. Using msp430-gdb, set break points at the start of each function
4. Set a watch for register 1 of the MSP430 microcontroller. Register 1 is used as the Stack Pointer (SP)
5. Run the program and keep track of the minimum value of SP. This is because the stack always grows up, thus the minimum value of SP would give us the maximum size of the stack
6. Subtract the minimum value of the stack from the value of SP at the `main()` function to get the stack depth of AES

Once we have a value for the maximum depth of the stack, we can add the size of the DATA segment and the BSS segment to account for initialized and uninitialized global variables.

5.2.3 Software Encryption Speed

is the number of bits of plaintext data that can be encrypted per second. Since the time to encrypt a single block of plaintext is on the order of microseconds, it is important to rely on a method that can measure at this resolution. To avoid any interference, we execute our code in a standalone mode without any underlying operating system on the msp430 and without the possibility of interruption.

We use the digital output pins of the MCU to set pins high just before initiating the encryption process and set it low just after completion. Using a oscilloscope capable of sampling voltage at a rate of 2 giga-samples per second, we recorded the square wave generated by the output pin going high and low and used the auto-measure feature of the scope to measure the time when the digital output pin remained high. We used an infinite loop which encrypted and decrypted a block of data. For AES, a single block is 128 bits in size. Our code sets the output pin high during encryption and low during decryption. The accuracy of this technique depends on the scale setting of the oscilloscope display. In Table 3, we list the scale settings that we have used for our measurements and the accuracy of each setting.

We also use this time measurement technique to measure time taken by the AES key expansion process and the CC2420 transmission rate.

Scale	Accuracy
$100\mu s/divsion$	$\pm 1\mu s$
$200\mu s/divsion$	$\pm 2\mu s$
$500\mu s/divsion$	$\pm 5\mu s$

Table 3: Scale settings and Accuracy of the Agilent DSO3202A

5.2.4 CC2420 Hardware Encryption Speed

poses a slightly different challenge because the encryption takes place on the CC2420 chip. We are limited by the interface it provides to the microcontroller to make any time measurements. Though the CC2420 supports a serial clock of up to 10 MHz, we are limited to 4 MHz by the MSP430 SPI (Serial Peripheral Interface) master mode. However, a serial clock of 4 MHz allows us to interact with the CC2420 at a rate of 4 Mbps which is much higher than the radio transmission rate of 250 kbps. Therefore, we do not see the serial link as a major performance bottleneck.

Similar to our timing method for software encryption, we used digital output pins of the MSP430 and an oscilloscope to measure time. Using the CC2420 hardware module involves multiple steps. These are:

1. Writing to the CC2420 RAM
2. Issuing the encrypt command to the CC2420
3. Wait for encryption module to complete processing by requesting status byte
4. Read from the CC2420 RAM

Though step 3 alone accounts for the time spent on encryption by the CC2420 hardware module, we need to factor in all of the steps listed above to get an application level estimate of encryption time.

Note that we evaluate the characteristics of the CC2420 in standalone encryption mode only. We assume that the circuitry used in standalone mode is the same as the circuitry used in inline mode. However, the CC2420 is not capable of performing decryption in standalone mode, so our results are limited to encryption only.

6. RESULTS

6.1 Effects of Optimizations

Fig. 5 shows the effects of applying our optimization techniques on performance, RAM and ROM with the O3 compiler option, Fig. 6 shows similar metrics without the O3 option.

6.1.1 Specialization of Code (SPECIAL)

Modifying the generic baseline implementation and making it specialized for AES-128 reduced the code size from 4942 to 4316 bytes. Due to the elimination of conditional constructs that accommodated the key expansion for different key sizes, we see a performance improvement of 183.52% in the key expansion process. Since the RAM size depends on the maximum depth of the stack, it is not effected by the key expansion process which happens before the encryption process that has a much greater stack requirement.

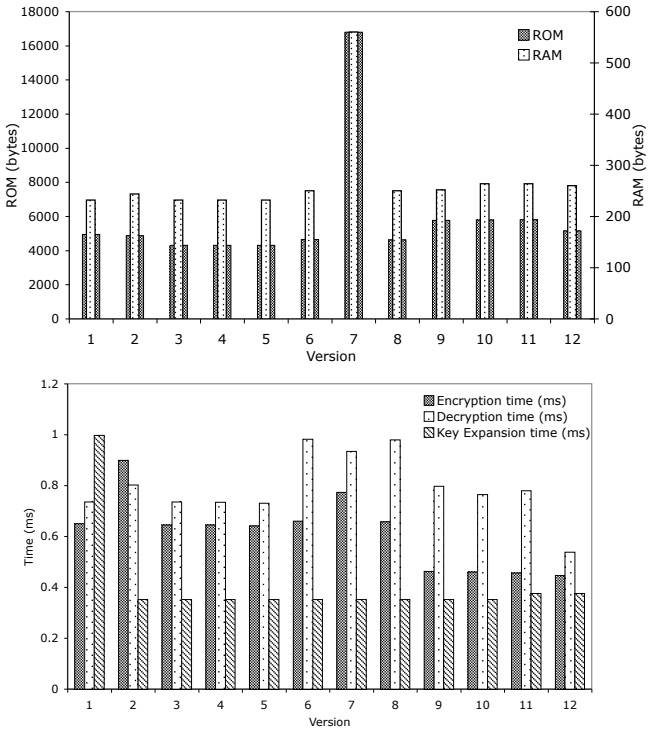

Figure 5: Effect of optimizations on encryptions speed, ROM and RAM usage. Refer to Table 4 for optimizations associated with version numbers

Version	Optimizations applied
1	NONE
2	SPECIAL, DATASZ(8-bits)
3	SPECIAL, DATASZ(16-bits)
4	SPECIAL, DATASZ(32-bits)
5	SPECIAL, DATASZ(64-bits)
6	SPECIAL, DATASZ(64-bits), MIX16
7	SPECIAL, DATASZ(64-bits), MIX16, UNROLL
8	SPECIAL, DATASZ(64-bits), MIX16, INLINE
9	SPECIAL, DATASZ(64-bits), MIX16, INLINE, REDMEM
10	SPECIAL, DATASZ(64-bits), MIX16, INLINE, REDMEM, LOCBUF
11	SPECIAL, DATASZ(64-bits), MIX16, INLINE, REDMEM, LOCBUF, GLOB
12	SPECIAL, DATASZ(64-bits), INLINE, REDMEM, LOCBUF, GLOB

Table 4: Optimizations associated with version number of each implementation. All versions compiled with -O3 option provided by msp-gcc

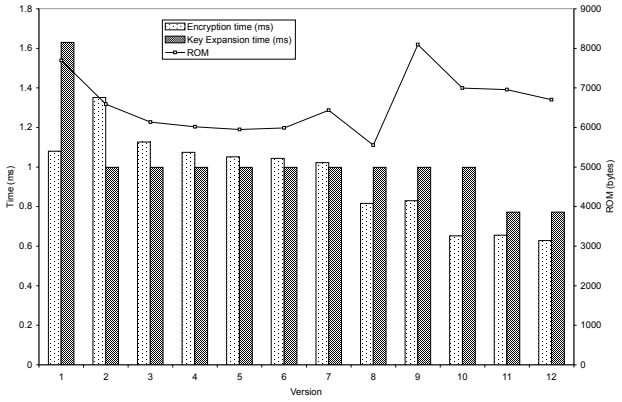

Figure 6: Effect of optimizations on encryption speed and code size (without the use of msp-gcc O3 option). Refer to Table 4 for optimizations associated with version numbers

6.1.2 *Varying Data Type Size (DATASZ)*

As expected, moving from 8-bit types to 16-bit types has a huge performance benefit of 39.53% due to the use of a 16-bit microcontroller. We also see a drop in code size from 4882 to 4314 bytes and RAM size from 244 to 232 bytes.

The mspgcc compiler also supports 32-bit and 64-bit types. On testing with these sizes, the gain in speed is negligible. However, on testing the same variations without the compiler optimization flag, we see a more noticeable difference. Without the O3 flag, in moving from 16-bit to 64-bit types, the speed increased by 7.03% while the ROM decreased from 6138 to 5950 bytes. This shows that the compiler optimizations work well to speed up the `AddRoundKey` transformation.

6.1.3 *Loop Unrolling (UNROLL)*

When the O3 compiler optimization is selected, the compiler automatically tries to perform loop unrolling as well as function inlining. We see that manually unrolling the loop when the O3 compiler optimization was selected had a negative impact on RAM, ROM and Speed. The RAM increased by more than 2.2 times and the code size increased by more than 3.61 times. The speed also decreased by 14.73%.

This effect of loop unrolling is counter-intuitive and is due to the compiler's inability to determine which portions of code need to be optimized. To verify this, we applied manual loop unrolling without the use of O3 and as expected, we observed a slight increase of 2.15% in speed and a 452 byte increase in code size.

6.1.4 *Function Inlining (INLINE)*

As mentioned before, O3 directs the compiler to attempt function inlining on the entire code. Therefore, compiling the source code with O3 with manual function inlining only gave us a negligible advantage. To evaluate the advantage gained due to function inlining, we compiled the source code without O3 and observed an increase of 20.55% in speed without sacrificing code space. This is because each transformation is called only once within the main loop which iterates through the rounds.

6.1.5 *Reducing Memory Moves (REDMEM)*

Reducing movement of data from one buffer to the other during the encryption process resulted in a 42.12% increase in performance. But this also increased the code size significantly by 1134 bytes.

When we tested the effects of reducing memory moves without the use of compiler optimizations, we observed a decrease in performance. This is because the compiler optimization enforces the use of more direct memory addressing which results in faster array accesses.

6.1.6 *Eliminate Local Function Buffers (LOCBUF)*

Use of a local buffer for the state within a function resulted in only a slight increase in performance, code size and RAM. This was again due to the effect of O3 which optimizes memory accesses using pointers. Without the use of O3, we see a more significant increase of 27.30% in performance and a decrease of 1096 bytes in code size.

6.1.7 Use of Global Variables (GLOB)

Storing the entire key schedule in a global variable hurt the performance of the key expansion process by 6.38% and resulted in a negligible improvement in encryption time. Again, without the use of O3, use of global key schedule improved key expansion performance significantly by 29.27%. This shows that the compiler is also effective at optimizing memory accesses for global variables.

6.1.8 On-the-fly-key Generation (OTFK)

Generating keys on-the-fly saves 160 bytes of RAM. This represents a key trade-off between performance and RAM usage. Performance is hurt only when encrypting multiple blocks as the round keys are recalculated for each block. This design choice largely depends on the size of the plaintext data to be encrypted using a single key.

6.1.9 16-bit Memory Writes in MixColumns (MIX16)

When compiled with the O3 option, using an 8-bit shift and or operation to generate a 16-bit value to write to RAM instead of writing two 8-bit values hurt performance by 2.22%. However, without the use of compiler optimizations, the performance showed a slight improvement. This shows that the compiler optimizes memory writes enough to make the use of 16-bit writes unnecessary.

6.2 Recommended Optimizations

Based on our analysis, we recommend these optimizations:

- SPECIAL
- DATASZ (64-bits)
- INLINE
- LOCBUF
- REDMEM
- GLOB

Using the msp430-gcc compiler at the O3 optimization level boosts performance by an additional 40.49%.

OTFK can be used on top of the above optimizations in cases where it is important to use minimal amount of RAM and the size of the plaintext data to be encrypted with a single key is not too large. If however, the data to be encrypted is large, then to prevent data replay attacks, different keys will have to be used anyway and therefore OTFK is less useful.

Using the above optimizations, we achieved an encryption speed of 286.35 kbps, RAM requirement of 260 bytes and a code size of 5160 bytes.

6.3 Comparison with Other Implementations

In Fig. 7, we see how our fastest implementation compares to previous attempts at optimizing AES on a similar platform. Our timing measurements for each implementation differ slightly from the published values as we compiled and tested each implementation on our platform. We can see that we have accomplished a significant improvement of 104.02% in encryption speed, 12.5% in key expansion and an overall improvement of 84.69% over the previous best performing implementation.

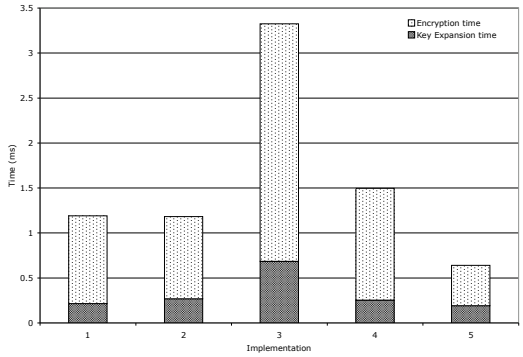

Figure 7: Comparison of Encryption Time + Key Expansion Time between our implementation (number 5) and other implementations (numbers 1-4). Refer to Table 5 for details about which implementation corresponds with which number above.

Additionally, Table 5 compares ROM usage among other implementations. Our version has the smallest ROM usage of all our empirically measured code sizes, 5160 bytes. Note that implementation 3 lists a smaller published ROM size, due most likely to our use of the -O3 compiler optimizations for our measured values.

RAM usage was similar among all implementations, and depends largely on whether keys are generated on the fly or pre-computed. Additionally, it is not clear how RAM usage was measured in other published implementations, especially with regard to stack usage.

Implementation	Reference paper	Measured ROM Usage	Published ROM Usage
1	[6]	5968 bytes	n/a
2	[12]	6780 bytes	12616 bytes
3	[14]	6848 bytes	3322 bytes
4	[10]	n/a	n/a
5	Our implementation	5160 bytes	n/a

Table 5: Information about implementations compared in Fig. 7. Measured ROM usage is taken from the reference implementation code we used, compiled with -O3 optimizations. Published ROM usage is taken directly from each published reference.

6.4 Comparison with Hardware Implementation

In this section we will examine the pros and cons of using a hardware implementation as well as describe our experience in getting it up and running.

Process	Time (μs)
Writing to the CC2420 RAM	94.40
Issuing the encrypt/decrypt command to the CC2420	6.40
Wait for encryption module to complete processing by requesting status byte	18.40
Read from the CC2420 RAM	102.40

Table 6: Time taken to complete each step required to encrypt using the CC2420 hardware AES module

Table 6 shows the time taken to encrypt using the CC2420 AES module. As expected, the hardware module is much faster than AES in software. This high speed of encryption

does not directly translate into a better performing WSN since the limiting factor of the network is the radio communication rate. We have already shown that AES in software can exceed the maximum specified rate of 250 kbps of IEEE 802.15.4-compliant WSNs. However, using the hardware module for data encryption does free up the microprocessor for a few milliseconds which can be used for other tasks. In a system where performance is crucial, allocating encryption to the CC2420 hardware and efficiently using microprocessor resources can result in slightly better performance.

The major disadvantage of using hardware AES is its lack of flexibility. Though AES-128 is sufficiently secure, security schemes are regularly evaluated and updated to ensure that they are not susceptible to newly developed attacks. In [8], NIST acknowledges that the widely used cipher-based authentication mode, CBC-MAC, has security deficiencies and details a specification for the CMAC mode. The CC2420 implements AES-based authentication using the CBC-MAC mode of operation, therefore we assume that its hardware implementation suffers from these deficiencies. Therefore, relying on hardware for security is a concern for secure sensor networks.

We also faced considerable difficulty in using the CC2420 AES hardware module due to lack of proper documentation. We found that the CC2420 is incapable of performing encryption/decryption unless the ciphertext is preceded by an 802.15.4-compliant header. This means that the hardware AES module cannot be used if the data are not formatted to be strictly compliant with IEEE 802.15.4. This is another major drawback since sensor networks almost always have different resource constraints that make it necessary to customize protocol specifications.

7. CONCLUSION

In this paper, we demonstrated that it is possible for an optimized C implementation of AES encryption-decryption to match the communication speed of a Zigbee radio. We show which optimizations work (and which do not) in increasing computational speed and reducing memory footprint. Additionally, we show how they interact with the optimizations of the GCC compiler. We provide a common, rigorous set of procedures and metrics for accurately measuring execution speed, ROM usage, and RAM usage. We use these metrics to benchmark our implementation along with four existing software implementations of AES on a common platform (Texas Instruments' MSP430 processor with a Chipcon CC2420 Zigbee radio) and show that our optimized implementation outperforms all previous implementations. We also evaluate the hardware implementation on the Zigbee radio and find that it outperforms all software-based schemes. However, this comes at the cost of lack of flexibility, e.g., different sized data blocks and difficulty of evolving to patch future security vulnerabilities.

In future work, we will be using the developed encryption-decryption scheme as a primitive in a secure application and to develop efficient interfaces with other primitives such as authentication. The ultimate goal is to provide an optimal, secure communication infrastructure for common embedded platforms.

8. REFERENCES

[1] Eyes project. `http://www.eyes.eu.org/`.

[2] Moteiv corportation. `http://www.moteiv.com/`.

[3] Softbaugh, inc. `http://www.softbaugh.com/`.

[4] *Advanced Encryption Standard (AES)*, FIPS PUB 197, November 2001.

[5] *IEEE Standard for Information technology- Telecommunications and information exchange between systems- Local and metropolitan area networks- Specific requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*, IEEE Standard 802.15.4-2006, September 2006.

[6] D.-R. Duh, T.-C. Lin, C.-H. Tung, and S.-J. Chan. An implementation of aes algorithm with the multiple spaces random key pre-distribution scheme on mote-kit 5040. In *SUTC '06: Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing - Vol 2 - Workshops*, pages 64–71, Washington, DC, USA, 2006. IEEE Computer Society.

[7] M. Dworkin. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. National Institute of Standards and Technology, December 2001.

[8] M. Dworkin. *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*. National Institute of Standards and Technology, May 2005.

[9] B. Gladman. Brian gladman's aes implementation. `http://fp.gladman.plus.com/AES/index.htm`.

[10] T. Instruments. Z-stack: Zigbee protocol stack from texas instruments, 2008.

[11] S. Karthikeyani. *IEEE 802.15.4TM and ZigBeeTM Hardware Platform using MSP430F1612*. Texas Instruments, September 2005.

[12] Y. W. Law, J. Doumen, and P. Hartel. Survey and benchmark of block ciphers for wireless sensor networks. *ACM Trans. Sen. Netw.*, 2(1):65–93, 2006.

[13] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[14] A. Vitaletti and G. Palombizio. Rijndael for sensor networks: Is speed the main issue? *Electron. Notes Theor. Comput. Sci.*, 171(1):71–81, 2007.