Fast Implementation of AES on Mobile Devices

Ashar Javed

Institute of Security in Distributed Applications
Hamburg University of Technology (TUHH)
Hamburg, Germany
ashar.javed@tu-harburg.de

Abstract: The Advanced Encryption Standard (AES) became the standard for encryption to protect the sensitive information. With the increasing use of portable and wireless devices and demanding information security needs in embedded systems, prompted efforts to find fast software based implementation of AES encryption/decryption capable of running on resource constrained environment in terms of processor speed, code space, energy usage and in particular those portable devices that have 32-bits ARM processor. ARM processor are most common for use in embedded industry. In this paper we propose an implementation of AES with minimum number of look-up tables in high level language C by performing a series of optimizations and their effects to enhance time performances that leads to our final implementation achieving speed of 323 μs to encrypt 128-bits block of plain text. We develop experiments by making the reference implementation, as known from the technical literature, optimized first for the 32-bits ARM based platforms and then compared with our final implementation. We analyze speed of AES, the leading symmetric block cipher on ARM processor and shows that our implementation outperforms the reference implementation by more than 3 times. The simulation results of our optimized implementation with the other reference implementation are compared and presented.

1 Introduction

Rijndael [DR00], an iterated block cipher algorithm designed by two Belgian cryptographers Vincent Rijmen and Joan Daemen, has been selected by NIST as the winner of the Advanced Encryption Standard competition among fifteen candidates due to its security, performance, efficiency, ease of implementation and flexibility. The AES developed to replace the old Data Encryption Standard (DES). According to NIST, DES algorithm can be easily broken in few hours with dedicated computer. Assuming that one could build a DES Cracker machine that could recover an unknown DES key in one second (i.e. try 255 keys per second), then it would take that machine approximately 149 thousand-billion (149 trillion) years to crack a 128-bits AES key [NQA]. To put that into perspective, the universe is believed to be less than 20 billion years old.

Rijndael is a symmetric block cipher with a variable block length and a variable key length. The block length and the key length can be independently specified to 128, 192 or 256-bits. The AES consider only 128-bits legal block length. The Rijndael cipher algorithm is

suited for an efficient implementation on a wide range of processors. AES is a substitution-permutation network in general sense. Each round of processing in AES involves a byte-level substitutions followed by byte level permutations. The nature of substitutions and permutations in AES allows efficient software implementations of the algorithm on wide range of platforms. The basic operations of the AES are simple. With the increasing use of portable and wireless devices in the business and daily life, protecting sensitive information via encryption is becoming more and more crucial. AES can be used as encryption standard in portable communication devices.

The contribution of this paper is fast software implementation of the AES algorithm in C with particular regard to 32-bits ARM [Ltd] processor. Efficient software implementations of AES proposed so far are listed in [BBF+02, DK06, ABM04]. However in this work to achieve faster speed than Version 1 (V1) [ABM04] we applied various optimizations techniques to Gladman's [Gla] AES implementation for low resource platforms. In this paper our work aims at comparing the V1 which is the faster encryption available according to [ABM04] on the target platform with our fastest implementation. For comprehensive comparison we also applied series of optimizations on different portions of the V1 on target platform.Now on we will call V1 as Newly Proposed V1. The time performances obtained by simulation are summarized in tables, compared and discussed. Our work covers the key setup and encryption / decryption of AES with 128-bits key size. This paper focuses only on key setup and encryption, but the method extends in natural way.

The rest of the paper is organized as follows. In the next section we give a short description of AES. Section 3 will describe the 32-bits ARM processor. In section 4, we shall discuss optimizations on Gladman's AES implementation for low resource platforms. Section 5 will describe the series of optimizations and their effects on V1. Section 6, will describe details of the time performances for comparison and tools used for implementation. In section 7, we summarize our results.

2 Description of AES Algorithm

This section provides a brief overview of the AES algorithm useful for understanding of the subsequent optimizations, which will be described in section 4 and 5. AES algorithm operates in rounds. A round is a fixed set of invertible transformations. The invertible transformations are also known as layers [BBF+02]. The transformations that are applied in each round are four. (1) round key addition step (2) the non linear step (3) the dispersion step (4) the diffusion step.They are described as follows.

2.1 AddRoundKey

In this step, the 128-bits of the state array are bitwise XOR'ed with the 128-bits of the round key. A core data structure of the algorithm is state array: it is 4*4 bytes matrix. Each 128-bits round key is generated from the key schedule.

2.2 SubBytes

This transformation is non linear byte-by-byte substitution. It operates independently on each byte of the state array. The goal of the substitution step is to reduce the correlation between input bits and output bits (at the byte level).

2.3 ShiftRows

The ShiftRows transformation consists of (i) not shifting the first row of the state array at all; (ii) circularly shifting the second row by one byte to the left; (iii) circularly shifting the third row by two bytes to the left; and (iv) circularly shifting the last row by three bytes to the left.

2.4 MixColumns

This transformation step replaces each byte of a column by a function of all the bytes in the same column. More precisely, each byte in a column is replaced by two times the value of that byte, plus three times the next byte, plus the byte that comes next, plus the byte that comes next. The word next means the byte in the row below; the meaning of next is circular in the same column. The transformation MixColumns requires matrix multiplication in the field GF(2^8).

3 ARM Processor

ARM [Ltd] is a 32-bits RISC microprocessor. ARM core is a high performance and low power consumption processor. The low power usage is particularly due to the fact that there is no cache, which can sacrifice performance. ARM is the industry standard for embedded system market. It covers 75 of the market with more than 1 billion sold every year. Their use in applications like digital radios, pagers, mp3 players and portable communication devices has necessitated software implementation of AES for the ARM. ARM is one of the most widely used 32-bits processors in mobile devices.

4 Optimizations on Gladman's Low Resource Implementation

In this section, we discuss important aspects of the Gladman's code [Gla] and the optimizations that we performed on Gladman's code to get a faster speed than Newly Proposed V1. We shall discuss in this section the reasons behind each optimization. We do not take any code-size or memory restrictions into account when designing the code for this platform.

Furthermore, we do not restrict ourself to any specific ARM processor, the code is portable among the ARM family of processors.

4.1 Use of Look-Up Tables

As mentioned in the AES proposal [DR], the algorithm can be considerably speed-up by precomputing the part of the internal operations and storing the results in the look-up tables. All the modular arithmetic of the AES can be reduced to a series of look-up tables and exclusive-or operations. Almost all the efficient implementations [BBF+02, DK06, ABM04] that we looked at take this approach of using look-up tables. In our case we want to speed up the algorithm with the minimum number of look-up tables. There are AES implementations that uses the four kilo bytes of look-up tables .The use of lot of look-up tables degrade the performance on the ARM processor.

ARM [Ltd] processor implements load / store architecture. Load / Store instructions can load or store 32-bits word or an 8-bits unsigned byte from memory to register or from register to memory. The data processing instructions manipulate data within registers. Most data processing instructions can process one of their operands using the barrel shifter. The duty of barrel shifter is to shift and rotate. Data processing instructions are processed within the arithmetic logic unit (ALU).

A unique and powerful feature of the ARM processor is the ability to shift the 32-bits binary pattern in one of the source registers left or right before it enters the ALU. Use of pre-rotated tables cannot improve the performance neither, since the barrel shifter that can be combined with data processing instructions reduces the effective cost of rotate instructions to zero. Use of such tables, in fact, increases the register pressure and possibility of cache misses therefore degrade the performance on ARM processor.It has no benefit in clock cycles. The look-up tables that we used in our case are three 256-bytes tables. One for S-Box, one for InvS-Box and one 256-bytes table for the construction of finite field elements.

4.2 Combinations 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 and by combining the round transformations very fast efficient implementation can be achieved . This technique was developed by Mark Malbrain and his contribution is acknowledged in Gladman's code. By making the parallelization of transformation we can speed up the process of encryption.

4.3 Tuning Options

Gladman's code has 3 options [DAB08] 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: This directs the compiler to take advantage of the memcpy function in the compiler's standard library. In our implementation we turned off this option due to bad repute of memcpy function regarding security features.

HAVE_UINT32: This directs the compiler to take advantage of 32-bits data types if available on the target platform. We use this feature in our implementation because ARM supports 32-bits data type.

VERSION_1: This makes extensive use of local buffers within functions instead of accessing data through pointers. Gladman said that the use of this option purely depends upon the targeting platforms regarding performance. In our implementation we have used this option.

4.4 Code Specialization

Gladman's code implements AES for 128 and 256-bits key sizes. In our implementation we specialize the code only for 128-bits key.

4.5 Change Data Type Size

One of the tuning options as mentioned earlier is to take advantage of the 32-bits data type. Gladman's code supports data types of 8 and 32-bits. ARM supports 32-bits so we use this option to get advantage. We observe a performance improvement with regards to speed. On ARM it does not make sense to use 8 bit data type because it degrades the performance.

4.6 Function Inlining and Use of Registers

It is a common optimization technique to speed up the performance. By making function inline we can avoid the function call and return overhead. It increases performance at the cost of code size. The reason behind this optimization is that all the portions of the AES algorithm are coded into single function. ARM supports inlining with the help of keyword __inline. In our implementation we make extensive use of registers in order to avoid memory moves to speed up the process of encryption. We kept the state array explicitly in registers.

4.7 Key Schedule

Round keys are derived from the process known as 'key scheduling'. There are two options to implement the key schedule: (1) key unrolling (2) On the fly key generation. The option we use in our implementation is key unrolling. We describe some reasons of not using the On the fly key generation approach. (a) Costly in clock cycles because key schedule have to performed for each data block to encrypt. (b) Need 16 bytes of additional memory to store the last round keys for the decryption. On the other way if you do not want to store the last 16 bytes of round keys, you need to calculate the key schedule again for the decryption and have to use it in reverse order.

5 Optimizations on Standard AES Algorithm

The original AES submission defines two implementation strategies. The first strategy which is known as V1 according to [ABM04]. V1 employs byte substitution tables only (S-Box and InvS-Box). V1 implements the nonlinear layer using byte substitution tables and implements the MixColumn transformation making repetitive calls to the Xtime operation. Xtime is a macro. V1 is also known as the compact version of the AES in technical literature. V1 is the fast encryption available on ARM platform.

We performed several optimizations on AES and key schedule. We also applied the standard techniques described in [Eff] to optimize the machine code generated by the ARM compiler. The main design goal is to optimize the implementation for speed.In this section we shall discuss the series of optimizations and the intuition behind them.

The compact version of the AES uses only two 256 bytes look-up tables. One for the S-BOX and other for the InvS-BOX. Our implementation also uses two 256 bytes look-up tables plus 10 bytes table for the round constant.The round constant is a word in which the three rightmost bytes are always 0. Thus the effect of an XOR of a word with round constant is to only perform an XOR on the leftmost byte of the word. The round constant is different for each round.

The S-BOX, which is invertible, is constructed by composing two transformations in $GF(2^8)$, an inversion and an affine function. One approach of implementing S-BOX is dynamically but it requires computation resources and it hurts the speed of the AES. Implementing S-BOX with the help of look- up tables is efficient in terms of time and space. In this implementation we restrict ourself with only two look-up tables plus 10 bytes of round constant. There are AES implementations that uses the four kilo bytes of look-up tables. As mentioned earlier that the use of lot of look-up tables degrade the performance on the ARM processor.

5.1 MixColumns

Different implementation strategies are available to implement this step. The compact version implements the MixColumns transformation making repetitive calls to the Xtime operation. Xtime is a macro. The multiplication with 02 can be implemented at byte level with a left shift and a conditional bitwise XOR with 1b. We restrict ourself by using this approach in our implementation because in this approach the execution time purely depends upon the input plain text. The other approach uses the log and anti-log table. We mentioned earlier that use of too many look-up tables degrade the performance on ARM processor [Ltd], and is considered unsafe, as monitoring the accesses to the look-up tables may in some cases allow to infer the key [BBF+02]. The transformation MixColumns requires matrix multiplication in the field GF(2^8).

5.2 ShiftRows

This transformation is straightforward to implement. The first row, row 0, is not rotated. Row 1 is rotated to the left by 1 byte position; row 2 is rotated to the left by 2 byte positions; row 3 is rotated to the left by 3 byte positions.

5.3 AddRoundKey

This transformation is also straightforward to implement. In this transformation, a round key is added to the State matrix by a simple bitwise XOR operation, that is, a sum in the field GF(2^8). Each round key is obtained from the key schedule.

5.4 KeyScheduling

The process of deriving a round key from the cipher key is known as key schedule. There are two options to implement the key schedule: (a) Key Unrolling (b) On the fly key generation. The option we use in our implementation is key unrolling. The reason for using the key unrolling is to speed up the time for making the key schedule. In our design we are focusing on speed and on the ARM enough memory is available to hold the unrolled key i.e. 176 bytes in our case of 128-bits AES encryption. The AES algorithm describes the calculation order of deriving the round keys from the cipher key in column wise fashion [DR00]. To speed up the calculation of deriving a round key from the cipher key,[LKCY03] proposed a method of calculating round keys in row wise fashion. By rearranging the calculation order we can save 50% of memory read write moves.
In our implementation we use this [LKCY03] approach of calculating the round keys to increase time performances. In our implementation we also use direct addressing method for doing index calculations to further boost the process of encryption. According to [Ltd],

16 out of 37, 32-bits registers are available at a time in ARM. In our implementation we make extensive use of registers in order to avoid memory moves to speed up the process of encryption.

6 Performance

This section illustrates our optimized version of AES algorithm that runs more than 3 times faster than Newly Proposed V1 implementation . We present results and the comparison with the Newly Proposed V1 implementation which uses the standard AES formulation combined with the row wise key scheduling technique [LKCY03]. In order to get the results found in tables below, we made a 128-bits key setup and encryption 100 times. The functions that we used for the calculation of the accurate timing are the QueryPerformanceCounter() and QueryPerformanceFrequency().To simulate a real world environment we started counting times before a call to a function performing the desired operation, such as encryption, decryption or key setup [DK06]. We stopped counting after the function was exited.

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 operations using fewer processor instructions. This ensures that our optimized implementations are in strict accordance with the AES specification. We verify the correctness of both implementations by comparing them with the test vectors included in [FIP01].

As the development software we used Microsoft Visual Studio 2008 with Microsoft Windows Mobile 6 Professional SDK. It uses the most popular ARM design ARM7TDMI. We used C language for the implementations.Table 1 shows the encryption speed of both methods when using 128-bits plain text block. Our final optimized implementation is more

Method	**Speed (min , avg) (μs)**
Our Final Implementation	(323 , 545) μs
Newly Proposed V1	(1200 , 3400) μs

Table 1: 128-Bits Encryption Speed.

than 3 times superior in speed than Newly Proposed V1 which is also optimized with regards to ARM as shown in Table 1. Table 2 shows the speed to calculate round keys with key unrolling [128-bits key] When calculating the key expansion, we found that our final

Method	**Speed (min , avg) (μs)**
Our Key Schedule (Column Wise Fashion)	(120 , 170) μs
Newly Proposed V1 (Row Wise Fashion)	(1590 , 4000) μs

Table 2: 128-Bits Key Schedule.

implementation outperforms the suggested key schedule process i.e. row wise calculation

[LKCY03] of Newly Proposed V1 by more than twenty times on average as shown in Table 2. In order to simulate the real world scenario, the scenario is to read data from the file that is stored on the disk and apply encryption with recommended mode of operations. The mode of operation we use in the implementations is cipher block chaining mode (CBC). Table 3 shows the result of encryption using CBC. By analyzing the Table 3, the result

	Our Final Implementation	Newly Proposed V1 Implementation
Size of Data (bytes)	**Speed(min , avg) '(ms)'**	**Speed(min , avg) '(ms)'**
16	(0.73, 1.07)	(1.20, 3.40)
32	(0.79, 1.17)	(2.53, 3.70)
64	(0.81, 1.25)	(4.70, 5.20)
128	(0.91, 1.53)	(5.58, 8.80)
256	(1.35, 1.90)	(10.91, 16.90)
512	(2.37, 2.65)	(20.93, 31.90)
1 Kilo Bytes	(3.74, 4.25)	(39.68, 60)
2 Kilo Bytes	(6.92, 7.70)	(88.28, 117)
4 Kilo Bytes	(11.51, 14)	(160, 201)
8 Kilo Bytes	(17.41, 27)	(315, 345)
16 Kilo Bytes	(34.06, 51)	(669, 690)
32 Kilo Bytes	(77.62, 89)	(1283, 1380)
64 Kilo Bytes	(134, 166)	(2529, 2710)
128 Kilo Bytes	(260, 290)	(5069, 5450)
256 Kilo Bytes	(533, 570)	(10130, 10584)
512 Kilo Bytes	(1053, 1090)	(20459, 20800)
1 Mega Bytes	(2117, 2170)	(42191, 44113)
2 Mega Bytes	(4318, 4442)	(83827, 87645)
4 Mega Bytes	(8374, 8650)	(161355, 193902)
8 Mega Bytes	(16748, 17528)	(347872, 393903)

Table 3: Encryption Speed Using CBC.

shows that our final implementation outperforms the Newly Proposed V1 for every data block size. More precisely for the data block of above or equal 1 MB our implementation is nearly 20 times faster than Newly Proposed V1.

7 Conclusion

An optimized version of the AES algorithm has been presented, coded in C and evaluated by simulations on ARM processor. We have rewritten the V1 by combining it with the new way of deriving a round keys that operates row wise, in order to minimize memory

moves. We have shown that our implementation outperforms the Newly Proposed V1 which is the fast encryption available on the target platform. We have also shown the comparison by considering the real life scenario in which user have the option to select file of any size ranging from 16 bytes to 8MB from the storage and perform encryption using CBC. Our final implementation preserves the flexibility of the AES and exhibits the highest encryption performance on ARM processor.

Some implementations perform a single encryption and /or decryption faster than other but fell behind for the bulk of encryption / decryption. It has been shown that our implementation works well in doing encryption either for one block of plain text or for 8MB of file. The ultimate goal is to provide an optimal secure storage and communication infrastructure on mobile devices.

References

[ABM04] Kubilay Atasu, Luca Breveglieri, and Marco Macchetti. Efficient AES Implementations for ARM Based Platforms. *Symposium on Applied Computing. Proceedings of the ACM symposium on Applied computing*, pages 841 – 845, 2004.

[BBF+02] G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti, and S. Marchesin. Efficient Software Implementation of AES on 32-Bit Platforms. *Proceedings of Cryptographic Hardware and Embedded Systems - CHES*, pages 129–142, 2002.

[DAB08] Shammi Didla, Aaron Ault, and Saurabh Bagchi. Optimizing AES for Embedded Devices and Wireless Sensor Networks. *Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks and communities.*, 2008.

[DK06] Matthew Darnall and Doug Kuhlman. AES Software Implementations on ARM7TDMI. *Proceedings of Progress in Cryptology - INDOCRYPT*, pages 424–435, 2006.

[DR] J. Daemen and V. Rijmen. AES Proposal: Rijndael. http://www.cryptosoft.de/docs/Rijndael.pdf (accessed 2010-05-05).

[DR00] J. Daemen and V. Rijmen. The Block Cipher Rijndael, 2000.

[Eff] Writing Efficient C for ARM. http://twins.ee.nctu.edu.tw/courses/ (accessed 2010-05-05).

[FIP01] Announcing the ADVANCED ENCRYPTION STANDARD (AES). Federal Information Processing Standard FIPS 197, National Institute of Standards and Technology (NIST), 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (accessed 2010-05-05).

[Gla] Brian Gladman's AES Implementation. http://www.gladman.me.uk/ (accessed 2010-05-05).

[LKCY03] Chi-Feng Lu, Yan-Shun Kao, Hsia-Ling Chiang, and Chung-Huang Yang. Fast Implementation of AES Cryptographic Algorithms in Smart Cards. 2003.

[Ltd] Arm Ltd. http://www.arm.com.

[NQA] NIST, AES Questions and Answers. http://csrc.nist.gov/archive/aes/index.html (accessed 2010-05-05).