FAST AES DECRYPTION

Vinit Azad
B.S., University of California, Davis, 2007

PROJECT

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

at

CALIFORNIA STATE UNIVERSITY, SACRAMENTO

SPRING
2011
Student: Vinit Azad

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the Project.

__________________________, Graduate Coordinator       ________________
Nikrouz Faroughi, Ph.D.                                        Date

Department of Computer Science
Abstract

of

FAST AES DECRYPTION

by

Vinit Azad

Due to the prevalent use of internet-enabled devices, confidentiality and privacy in communication has been more important than ever. Encryption algorithms, such as AES and RSA, are used to ensure the security and privacy of such communications. However, as computers get computationally powerful, more complex encryption algorithms are needed to avoid brute force attacks. This complexity in the algorithms also means that encryption and decryption of messages using these algorithms can be slow. To avoid this decrease in speed, many optimizations have been proposed to make these complex algorithms perform much faster.

Emilia Käsper and Peter Schwabe proposed one such optimization to AES in their paper “Faster and Timing-Attack Resistant AES-GCM.” They describe a fast constant-time AES encryption algorithm, which is also immune from cache-timing attacks, using Intel’s SSE instructions. This paper focuses on the study of Käsper and Schwabe’s proposal and implements a fast constant-time AES decryption algorithm that can work with their encryption algorithm. The paper concludes that while the decryption algorithm is slower than the encryption algorithm proposed by Käsper and Schwabe, it is about 3 times faster than the standard decryption implementation such as OpenSSL. Therefore, this fast decryption algorithm paired with Käsper and Schwabe’s fast encryption
algorithm can provide users a complete AES package that can be used in applications that require fast encryptions and decryption along with protection against timing attacks.

_______________________, Committee Chair
Ted Krovetz, Ph.D.

_______________________
Date
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td></td>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>1.</td>
<td>INTRODUCTION TO AES</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>AES State</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>AES Encryption Algorithm</td>
<td>2</td>
</tr>
<tr>
<td>1.2.1</td>
<td>ShiftRows</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2</td>
<td>MixColumns</td>
<td>4</td>
</tr>
<tr>
<td>1.2.3</td>
<td>AddRoundKey</td>
<td>4</td>
</tr>
<tr>
<td>1.2.4</td>
<td>SubBytes</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>AES Decryption Algorithm</td>
<td>6</td>
</tr>
<tr>
<td>1.3.1</td>
<td>InvShiftRows</td>
<td>8</td>
</tr>
<tr>
<td>1.3.2</td>
<td>InvMixColumns</td>
<td>8</td>
</tr>
<tr>
<td>1.3.3</td>
<td>InvSubBytes</td>
<td>9</td>
</tr>
<tr>
<td>1.4</td>
<td>Key Expansion Algorithm</td>
<td>9</td>
</tr>
<tr>
<td>2.</td>
<td>BITSLICED AES ENCRYPTION</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>Bitslicing</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Bitsliced ShiftRows</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Bitsliced MixColumns</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Bitsliced AddRoundKey</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>Bitsliced SubBytes</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>Bitsliced Key Expansion Algorithm</td>
<td>18</td>
</tr>
<tr>
<td>3.</td>
<td>BITSLICED AES DECRYPTION</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>Bitsliced InvShiftRows</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Bitsliced InvMixColumns</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Bitsliced AddRoundKey</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>Bitsliced InvSubBytes</td>
<td>25</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>3.5 Bitsliced Inverse Key Expansion Algorithm</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>4. VALIDATION AND PERFORMANCE ANALYSIS</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>5. CONCLUSION</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Appendix A. Bitsliced Decryption Code</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>Appendix B. OpenSSL Decryption Code</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Appendix C. System Configuration</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>References</td>
<td>71</td>
<td></td>
</tr>
</tbody>
</table>
LIST OF TABLES

1. Table 4.1: Performance data for Core2Duo CPU ....................................................... 28
2. Table 4.2: Instruction count for AES operations .......................................................... 29
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Figure 1.1: Operation performed on State array</td>
<td>2</td>
</tr>
<tr>
<td>2.</td>
<td>Figure 1.2: AES encryption pseudocode</td>
<td>3</td>
</tr>
<tr>
<td>3.</td>
<td>Figure 1.3: ShiftRows transformation</td>
<td>4</td>
</tr>
<tr>
<td>4.</td>
<td>Figure 1.4: Column multiplication</td>
<td>4</td>
</tr>
<tr>
<td>5.</td>
<td>Figure 1.5: AES decryption pseudocode</td>
<td>7</td>
</tr>
<tr>
<td>6.</td>
<td>Figure 1.6: InvShiftRows transformation</td>
<td>8</td>
</tr>
<tr>
<td>7.</td>
<td>Figure 1.7: Inverse column multiplication</td>
<td>8</td>
</tr>
<tr>
<td>8.</td>
<td>Figure 1.8: Key expansion algorithm</td>
<td>10</td>
</tr>
<tr>
<td>9.</td>
<td>Figure 2.1: Bitsliced 8 AES-State</td>
<td>13</td>
</tr>
<tr>
<td>10.</td>
<td>Figure 2.2: ShiftRows using pshufb</td>
<td>14</td>
</tr>
<tr>
<td>11.</td>
<td>Figure 2.3: Equation for bitsliced MixColumns</td>
<td>15</td>
</tr>
<tr>
<td>12.</td>
<td>Figure 2.4: Matrix used to change from standard basis to subfield basis</td>
<td>17</td>
</tr>
<tr>
<td>13.</td>
<td>Figure 2.5: Subfield basis to standard basis along with affine transformation</td>
<td>18</td>
</tr>
<tr>
<td>14.</td>
<td>Figure 3.1: InvShiftRows using pshufb</td>
<td>20</td>
</tr>
<tr>
<td>15.</td>
<td>Figure 3.2: Multiplication using shifts and XOR</td>
<td>20</td>
</tr>
<tr>
<td>16.</td>
<td>Figure 3.3: Shifts using XOR</td>
<td>22</td>
</tr>
<tr>
<td>17.</td>
<td>Figure 3.4: Galois multiplication using XORs</td>
<td>23</td>
</tr>
<tr>
<td>18.</td>
<td>Figure 3.5: InvMixColumns equations</td>
<td>24</td>
</tr>
<tr>
<td>19.</td>
<td>Figure 3.6: Inverse affine transform and standard to subfield basis change</td>
<td>25</td>
</tr>
<tr>
<td>20.</td>
<td>Figure 3.7: Subfield to standard basis change matrix</td>
<td>25</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION TO AES

Due to prevalent use of internet-enabled devices, confidentiality and privacy in communication has been more important than ever. To provide this confidentiality for electronic communications, encryption algorithms are used \[7\]. There are two kinds of encryption algorithms: symmetric-key encryption and public-key encryption \[7\].

Symmetric encryption is also known as a single key encryption \[7\] because a single key is used for both encryption and decryption algorithm. Public-key encryption algorithm is an algorithm where a public key (available to all) and a separate private key (only known to one entity) are used for encryption and decryption \[7\]. Data Encryption Standard (DES), Advanced Encryption Standard (AES), Blowfish, and Serpent are all examples of symmetric key algorithm \[7\]. This paper will focus only focus on symmetric encryption, more specifically on AES.

The AES algorithm, also known as Rijndael algorithm, is a symmetric block cipher that can encrypt blocks of 128-bit data \[1\]. It can use 128, 192, or 256 bits of key length. The algorithm, created to replace DES, was adopted by NIST (National Institute of Standards and Technology) as a U.S. government standard in 2001 \[6\]. The AES algorithm involves several number of rounds with each round performing certain operations on the input to produce the output \[1\]. The number of rounds performed depends on the key length \[1\]. The operations performed in each round are called ShiftRows, MixColumns, AddRoundKey and SubBytes \[1\]. The following sections in this chapter will go into AES concepts and description of each operation performed in a
round. The full AES encryption and decryption algorithm is explained in the AES encryption and decryption sections, respectively.

1.1 AES State

The AES algorithm performs each operation on a two-dimensional array, called the State [1]. The State consists of 4 bytes of rows with 4 bytes of columns [1]. At the beginning of the algorithm, the input is copied into the State [1]. The 4 AES operations mentioned previously are performed on the State and copied into the output array, as seen in Figure 1.1[1].

![Figure 1.1: Operation performed on State array][1]

1.2 AES Encryption Algorithm

The AES algorithm specified by NIST in the AES specification is shown in Figure 1.2. Nb refers to the number of columns in the AES state. NIST’s AES specification specifies this number as 4. Nr refers to the number of rounds, which is a function of the key size. Nr is 10 if the key size is 128 bits, 12 if the key size is 196 bits, or 14 if the key size is 256 bits [1].
The algorithm starts out by copying the input into the state, followed by an addition of the round key to the state. Round key calculation is explained in the key expansion section. Next, the algorithm performs \( N_r - 1 \) rounds of the following operations: SubBytes, ShiftRows, MixColumns and AddRoundKey. The final round is the same as other rounds except that MixColumns operation is not performed. The algorithm ends after copying the state to the output.

1.2.1 ShiftRows

The ShiftRows operation performs a left shift of the 3 bottom rows in the State array [1]. Each row is shifted by a different amount [1]. The first row is not shifted, the second row is shifted left by 1, the third row is shifted left by 2, and the fourth row is shifted left by 3 [1]. This operation is shown in Figure 1.3.

Figure 1.2: AES encryption pseudocode [1]
1.2.2 MixColumns

The MixColumns operation works on the columns of the State array [1]. The columns of the State is multiplied using the matrix shown in Figure 1.4 [1]. The multiplication is done in Galois Field GF(2) modulo a irreducible polynomial [1]. For AES, this polynomial corresponds to 0x11b when translated to hex [1].

\[
\begin{bmatrix}
S'_{0,c} \\
S'_{1,c} \\
S'_{2,c} \\
S'_{3,c}
\end{bmatrix} =
\begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02
\end{bmatrix}
\begin{bmatrix}
S_{0,c} \\
S_{1,c} \\
S_{2,c} \\
S_{3,c}
\end{bmatrix}
\quad \text{for } 0 \leq c < Nb.
\]

Figure 1.4: Column multiplication [1]

1.2.3 AddRoundKey

The AddRoundKey operation adds the round key to the AES State [1]. An addition operation in the AES algorithm is defined as just a bitwise XOR [1]. The key is
added to the columns using the formula below, where round is the number of the round currently being performed and C is the column number in the AES State [1]:

\[
[S'_{0,C}, S'_{1,C}, S'_{2,C}, S'_{3,C}] = [S_{0,c}, S_{1,c}, S_{2,c}, S_{3,c}] \oplus [\text{Key}_{\text{round}*4+C}].
\]

1.2.4 SubBytes

The SubBytes operation transforms each byte of the State independently using a non-linear substitution table called the S-box [1]. According to the AES specification, the S-box is created by applying the following transformations:

1. Take the multiplicative inverse of the byte in AES State in GF(2^8), where byte 0x00 is mapped to itself.

2. Apply the affine transformation shown in matrix form below:

\[
\begin{bmatrix}
    b_0 \\
    b_1 \\
    b_2 \\
    b_3 \\
    b_4 \\
    b_5 \\
    b_6 \\
    b_7
\end{bmatrix}
\begin{bmatrix}
    1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
    1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
    1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
    1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
    0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
    0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
    0 & 0 & 0 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
    b_0 \\
    b_1 \\
    b_2 \\
    b_3 \\
    b_4 \\
    b_5 \\
    b_6 \\
    b_7
\end{bmatrix}
\]

Calculating the multiplicative inverse of an element in Galois Field GF(2^8) is not simple and therefore, many of current AES implementation use a lookup table to map the input to the output [2].
1.3 AES Decryption Algorithm

The decryption algorithm specified by NIST is shown in Figure 1.5. This “straightforward implementation” is implemented by reversing the encryption algorithm and inverting the transformations [1]. However, NIST specifies another equivalent decryption algorithm that does not require rearranging the operations within the algorithm [1]. Therefore, this equivalent decryption algorithm will have the operations specified in the same order as the encryption algorithm except the operations are the inverse of the encryption operations. However, this algorithm requires a change in the key expansion algorithm while the straightforward algorithm specified in Figure 1.5 can use the same key expansion algorithm for both encryption and decryption. The equivalent algorithm provides a much better structure between the encryption and decryption since the operations are not rearranged [1]. The structure of the decryption algorithm implemented in this paper does not borrow any structure from the encryption algorithm. Therefore, to avoid changes to the key expansion function, this paper will focus on the straightforward decryption algorithm specified by NIST.
The decryption algorithm involves additional operations: InvShiftRows, InvSubBytes, and InvMixColumns. These operations are the inverse of the ShiftRows, SubBytes and MixColumns operations, respectively [1]. AddRoundKey does not have an InvAddRoundKey operation because AddRoundKey operation performs a round key addition using XOR. Since XOR is its own inverse, AddRoundKey is also its own inverse.

Similar to encryption algorithm, the decryption algorithm starts by copying the input into the state followed by the round key addition. The Nr-1 rounds are performed by applying the following operations to the AES state: InvShiftRows, InvSubBytes, AddRoundKey, and InvMixColumns. The final round is similar to other rounds except it does not perform the InvMixColumns operation. The algorithm finishes by copying the state into the output.
1.3.1 InvShiftRows

InvShiftRows is the inverse of ShiftRows operation. Since ShiftRows operation shifts the last 3 rows of the state left by a specific amount, InvShiftRows shifts the last 3 rows of the state right by the same amount to revert the changes of ShiftRows [1]. This operation is shown in Figure 1.6.

![Figure 1.6: InvShiftRows transformation [1]](image)

1.3.2 InvMixColumns

InvMixColumns is the inverse of MixColumns operation. NIST provides the inverse of the GF(2⁸) polynomial used in the MixColumns operation. This inverse polynomial is shown in its matrix form in Figure 1.7. As with MixColumns operation, the multiplication in InvMixColumns is also done module 0x11b.

\[
\begin{bmatrix}
S_{0,c} \\
S_{1,c} \\
S_{2,c} \\
S_{3,c}
\end{bmatrix} = \begin{bmatrix}
0e & 0b & 0d & 09 \\
09 & 0e & 0b & 0d \\
0d & 09 & 0e & 0b \\
0b & 0d & 09 & 0e
\end{bmatrix} \begin{bmatrix}
S_{0,c} \\
S_{1,c} \\
S_{2,c} \\
S_{3,c}
\end{bmatrix} \quad \text{for } 0 \leq c < Nb
\]

![Figure 1.7: Inverse column multiplication [1]](image)
1.3.3 InvSubBytes

InvSubBytes is the inverse of SubBytes operation. The inverse is calculated by applying the inverse affine transformation followed by taking the multiplicative inverse in GF($2^8$) [1]. Once again, since the calculating the multiplicative inverse is not simple, many implementation use the inverse S-box table to calculate the InvSubBytes operation.

1.4 Key Expansion Algorithm

The key expansion algorithm takes the 128 bit, 196 bit, or 256 bit key and expands it to $4 \times (N_r + 1)$ words, where $N_r$ is the number of rounds, to use as round key in the encryption and decryption algorithm [1]. The key expansion algorithm is shown in Figure 1.8.
Figure 1.8: Key expansion algorithm [1]

The algorithm specifies the key expansion for all key sizes by including the variable Nk. Nk is the number of 32-bit words in the original key [1]. However, this paper will focus only on 128-bit keys. Therefore, for this paper, Nk is defined as 4. Since Nk is 4, the “else if” statement in the while-loop will never execute and can be removed. When using the straightforward AES decryption algorithm, this key expansion algorithm will produce the round key for both encryption and decryption [1].

The Key expansion algorithm starts by copying the user provided key into the first 16 bytes of the round key. The key expansion algorithm uses SubWord, RotWord, Rcon, and the previous byte of the round key to calculate new round keys. The SubWord operation is simply the SubBytes operation performed on each byte of the word [1]. The
RotWord operation rotates the word left by 32 bits [1]. Rcon is constant word array containing the value \( \{2^{i-1}, 0, 0, 0\} \), where \( i \) starts at 1 [1].
Chapter 2

BITSLICED AES ENCRYPTION

In their paper, “Faster and Timing-Attack Resistant AES-GCM,” Emilia Käsper and Peter Schwabe present a constant-time bitsliced implementation of AES encryption for 64-bit Intel processors. The implementation is done in assembly language using SSE instructions. It takes advantage of the 16 128-bit XMM registers and SSE3 instructions available in Intel processors. Their implementation focuses on 128-bit keys to take full advantage of the 128-bit registers. Since most of the SSE instructions use 2 XMM registers, 8 XMM registers are used as input while the other 8 XMM registers are used as output.

Käsper and Schwabe’s implementation performs at 7.59 cycles/byte on an Intel Core 2 Quad CPU, which is 25% faster than previous implementations [5]. On an Intel Core i7 CPU, the same implementation performs at 6.92 cycles/byte [Kaper]. Since the implementation is a constant-time implementation, it also prevents against cache-timing attacks.

Cache-timing attacks are attacks that exploit the amount of time it takes to load a data from memory [5]. A data load from the cache will have a smaller load time than a data load from the RAM [5]. Since many AES implementations use a pre-computed lookup table to implement an S-box, AES is vulnerable to cache-timing attacks [4]. A simple attack proposed by Bernstein in [4] proposes that the access time for S-box table depends on the array index. Since the array index is determined using the AES state, the table lookup timing data can be used to determine information about the AES state [4].
However, if the S-box is computed using a constant-time implementation instead of a lookup table, then this cache timing attack can be avoided. Käsper and Schwabe’s AES implementation uses a constant time S-box implementation proposed by Canright in [3] to avoid the cache-timing attacks.

2.1 Bitslicing

Bitslicing is a technique that simulates hardware implementation in software [5]. Käsper and Schwabe’s try to follow hardware implementation, such as for S-box, of AES as much as possible to achieve the constant-time implementation. Since AES works on a 4x4 AES state, it would be too inefficient to perform operation on those 16 bytes separately. Therefore, Käsper and Schwabe proposed that individual byte be split into 8 XMM register as shown in Figure 2.1.

![Figure 2.1: Bitsliced 8 AES-State](image)

Bit 0 of a byte will be stored in XMM0, Bit 1 of a byte will be stored in XMM1, and so on until Bit 7 is stored in XMM7. Since each XMM register can store 128 bits, 128 bytes can be stored in 8 XMM registers in such a way. By performing the same instruction on all 8 XMM registers, the instruction will be applied to 128 bytes together. Processing
128 bytes means 8-blocks of AES can be processed at once in counter mode. The bitsliced version of each AES operation will be discussed in the following sections.

2.2 Bitsliced ShiftRows

Käsper and Schwabe use the SSE3 instruction called pshufb to implement ShiftRows. This instruction shuffles each byte in the register using a mask. Since each byte of the bitsliced state contains 8 bits from 8-AES state, the ShiftRows can be implemented using the permutation pattern shown in Figure 2.2.

\[
\begin{bmatrix}
    a_0 | a_1 | a_2 | a_3 | a_10 | a_11 | a_12 | a_13 | a_20 | a_21 | a_22 | a_23 | a_30 | a_31 | a_32 | a_33
\end{bmatrix} \mapsto \\
\begin{bmatrix}
    a_0 | a_1 | a_2 | a_3 | a_10 | a_11 | a_12 | a_13 | a_20 | a_21 | a_22 | a_23 | a_30 | a_31 | a_32 | a_33
\end{bmatrix}
\]

Figure 2.2: ShiftRows using pshufb [5]

Using pshufb, the ShiftRows operation can be implemented using one instruction per XMM register. Core i7 adds an additional pshufb unit to the processor. Therefore, on a Core i7, two pshufb instructions can be submitted together increasing the total throughput [5].

2.3 Bitsliced MixColumns

MixColumns multiplies the AES state with a fixed matrix to obtain a new state, followed by a conditional masking with 0x1b whenever the most significant bit is 1 [1]. The most significant bit is checked because the mod is only applied when the value exceeds 8-bits. If the most significant bit is 1, then the value will exceed 8-bits after a left shift.
The fixed matrix only contains 0x01, 0x02, or 0x03. Therefore, the multiplication can be implemented using shifts and additions. The trivial case of multiplication with 0x01 results in the same output as the input. A multiplication with 0x02 can be implemented by left shifting the multiplicand once. Finally, a multiplication with 0x03 can be implemented by left shifting the multiplicand once, then adding the multiplicand to the result. Recall that the additions in AES operations are just XORs.

Since after bitslicing the AES state, each register contain 1 bit of the each byte in the state, bitsliced MixColumns will work on 1 bit of the byte at a time. Therefore, the left shift is implemented by just using the bit immediately to the right in the calculations. The conditional XOR with 0x1b is implemented by XORing the result with bit 7. If bit 7 is 1, then the XOR will be performed correctly. If bit 7 is 0, then the XOR will have no affect on the result. The equations to calculate the bitsliced MixColumns are shown in Figure 2.3.

\[
\begin{align*}
    b_{ij}[0] &= a_{ij}[7] \oplus a_{i+1,j}[0] \oplus a_{i+1,j}[7] \oplus a_{i+2,j}[0] \oplus a_{i+3,j}[0] \\
    b_{ij}[1] &= a_{ij}[0] \oplus a_{ij}[7] \oplus a_{i+1,j}[0] \oplus a_{i+1,j}[1] \oplus a_{i+1,j}[7] \oplus a_{i+2,j}[1] \oplus a_{i+3,j}[1] \\
    b_{ij}[2] &= a_{ij}[1] \oplus a_{i+1,j}[1] \oplus a_{i+1,j}[2] \oplus a_{i+2,j}[2] \oplus a_{i+3,j}[2] \\
    b_{ij}[3] &= a_{ij}[2] \oplus a_{ij}[7] \oplus a_{i+1,j}[2] \oplus a_{i+1,j}[3] \oplus a_{i+1,j}[7] \oplus a_{i+2,j}[3] \oplus a_{i+3,j}[3] \\
    b_{ij}[4] &= a_{ij}[3] \oplus a_{ij}[7] \oplus a_{i+1,j}[3] \oplus a_{i+1,j}[4] \oplus a_{i+1,j}[7] \oplus a_{i+2,j}[4] \oplus a_{i+3,j}[4] \\
    b_{ij}[5] &= a_{ij}[4] \oplus a_{i+1,j}[4] \oplus a_{i+1,j}[5] \oplus a_{i+2,j}[5] \oplus a_{i+3,j}[5] \\
    b_{ij}[6] &= a_{ij}[5] \oplus a_{i+1,j}[5] \oplus a_{i+1,j}[6] \oplus a_{i+2,j}[6] \oplus a_{i+3,j}[6] \\
    b_{ij}[7] &= a_{ij}[6] \oplus a_{i+1,j}[6] \oplus a_{i+1,j}[7] \oplus a_{i+2,j}[7] \oplus a_{i+3,j}[7].
\end{align*}
\]

*Figure 2.3: Equation for bitsliced MixColumns [5]*
2.4 Bitsliced AddRoundKey

The bitsliced version of AddRoundKey is implemented simply by XORing the bitsliced version of the round key to the bitsliced AES state [5]. To improve the performance of bitsliced AddRoundKey, the key addition is interleaved with the pshufb instructions from the ShiftRows operation [5].

2.5 Bitsliced SubBytes

Käsper and Schwabe use a hardware implementation by Canright [3] to turn S-box into Boolean instructions [5]. Since calculating the inverse in GF($2^8$) directly is difficult, the 8-bit calculations can be expressed in terms of 4-bit calculations using subfield arithmetic and 4-bit calculations can be expressed in terms of 2-bit calculations [3]. The inverse is calculated using 2-bits, which gives the result for 4-bit, which gives the result for the original 8-bit problem [3]. This paper will not go into details of the S-box hardware algorithm implementation but the representation used in the S-box hardware algorithm will be briefly discussed.

The S-box algorithm involves several representations (8-bit data represented in 4-bit data, etc) in Galois Field [3]. Therefore, each byte of data needs to be converted into a new representation (subfield basis), go through the S-box algorithm, and then converted back to its original representation (standard basis) [3]. This change of basis can be achieved using a matrix multiplication [3].

For encryption, an affine transformation need to be performed after calculating the multiplicative inverse [1]. Since the affine transformation can be calculated using a
matrix multiplication, the affine transformation matrix and the change of basis (from subfield basis to standard basis) can be combined into one matrix to avoid multiple matrix multiplication [3].

Therefore, the SubBytes operation for encryption can be calculated by changing the byte from standard basis to subfield basis by multiplying the byte using matrix $X^{-1}$ shown in Figure 2.4 [3]. After the change of basis, the multiplicative inverse can be calculated using the S-box algorithm [3]. The multiplicative inverse can now be changed back to standard basis and the affine transformation can be applied all together by multiplying the inverse using matrix $MX$ shown in Figure 2.5 [3]. The S-box calculation can be finished by adding the constant specified in the AES specification [3].

$$\begin{pmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} g_7 \\ g_6 \\ g_5 \\ g_4 \\ g_3 \\ g_2 \\ g_1 \\ g_0 \end{pmatrix}$$

Figure 2.4: Matrix used to change from standard basis to subfield basis [3]
2.6 Bitsliced Key Expansion Algorithm

The Bitsliced key expansion algorithm produces bitsliced round keys that can be used directly with the bitsliced state without any transformation. The key expansion algorithm uses bitsliced SubBytes operation to implement the algorithm. To avoid doing addition with a constant that is needed in the SubBytes operation, the constant is added directly to the round keys [5]. This optimization will save 4 XOR operation during each SubBytes operation.
Chapter 3
BITSLICED AES DECRYPTION

This chapter will go into the details of implementing the decryption algorithm associated with the Käsper and Schwabe’s encryption algorithm. The decryption algorithm is based on Käsper and Schwabe’s implementation, therefore their code is modified to support decryption. The original code is available on Emila Käsper’s website. The decryption code and any modified code from Käsper’s original implementation is shown in Appendix A.

The decryption algorithm follows the organization recommended by NIST and described in Section 1.3. The rest of the chapter will describe the bitsliced implementation of the individual operations required for decryption.

3.1 Bitsliced InvShiftRows

As described in Section 1.3.1, InvShiftRows operation shifts the last 3 rows of the state to the right by different amounts. As with the bitsliced ShiftRows operation, pshufb instruction is used to shift the bits in correct position. Figure 3.1 shows how the InvShiftRows operation modifies the state using the pshufb instruction in the XMM register. Käsper and Schwabe interleaved ShiftRows and AddRoundKey in their encryption implementation; however, those operations cannot be interleaved due to the different order of operation in the decryption algorithm. Therefore, the decryption algorithm separates out the InvShiftRows and AddRoundKey to two different operations.
3.2 Bitsliced InvMixColumns

As explained in Section 1.3.2, InvMixColumns is implemented by multiplying the state with the inverse polynomial associated with the MixColumns operation. MixColumns operation only consisted of multiplication with 2 and 3 which could be implemented simply by shifts and XOR. Same idea can be used to implement InvMixColumns, however InvMixColumns consists of multiplication with 0x9, 0xb, 0xc, and 0xd, which will require a lot more shifts and XORs than the MixColumns operation. The equations used to multiply by 0x9, 0xb, 0xc, and 0xd using shifts and XOR is shown in Figure 3.2.

\[
\begin{align*}
09 &= x << 3 \oplus x \\
0b &= x << 3 \oplus x << 1 \oplus x \\
0d &= x << 3 \oplus x << 2 \oplus x \\
0e &= x << 3 \oplus x << 2 \oplus x << 1
\end{align*}
\]

Figure 3.2: Multiplication using shifts and XOR

When multiplying in Galois Field, one needs to remember to XOR with 0x11b if bit 7 was 1 before the shift. Therefore, after each bit-shift, an XOR might be needed if bit 7 was set before the shift.

The AES state has already been bitsliced by the time this operation will be performed. Therefore, the bit shifts can be implemented statically instead of using the
bitshift operation. In addition, since after each bit shift, there might be a need for a XOR by 0x11b, this bit shift cannot be implemented using the bit shift instruction. The conditional XOR after the bit shift can be implemented statically too. If bit 7 were checked before deciding whether to perform a XOR, it would make this implementation vulnerable to timing attacks. Therefore, to preserve the constant timing of the algorithm, all the operations are converted into simple XORs.

For example, left shifting bit 1 can be implemented by realizing that after the shift bit 0 will be in position of bit 1 and if bit 7 of the bit was set, then bit 1 will need to be XORed by 1 (since the whole byte will be XORed by 0x11b). Therefore, a left shift of bit1 can be achieved using the following equation: $x[1] = x[0] \oplus x[7]$. Similar idea can be applied to obtain equation for left shift by 2 and 3. The equation for all the shifts used in Figure 3.2 is shown in Figure 3.3. Using these equations, all the shifts and modulo can be implemented just using XORs.
Using equation from Figure 3.2 and equations from 3.3, equations can be created to multiply by 0x9, 0xb, 0xd, and 0xe. These equations are shown in Figure 3.4.
Using equations from Figure 3.4, the InvMixColumns operation can be implemented.

Due to matrix multiplication’s symmetry, the matrix multiplication can be written as a generic equation: $b_{ij} = 0e*a_{ij} \oplus 0b*a_{i+1,j} \oplus 0d*a_{i+2,j} \oplus 09*a_{i+3,j}$, where the row number $i$ is modulo 4 [5]. The state is stored row-by-row in the XMM registers. Therefore, the $a_{i+1}$ can be obtained by left shifting by 32 (i.e, 1 row), $a_{i+2}$ can be obtained by left shifting by 64, and $a_{i+3}$ can be obtained by left shifting by 92. The equations to perform the whole InvMixColumns are shown in Figure 3.5.

### Figure 3.4: Galois multiplication using XORs

<table>
<thead>
<tr>
<th>Multiplication by 09</th>
<th>Multiplication by 0b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x'[0] = x[5] \oplus x[0]$</td>
<td>$x'[0] = x[5] \oplus x[7] \oplus x[0]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multiplication by 0d</th>
<th>Multiplication by 0c</th>
</tr>
</thead>
</table>
\[
\begin{align*}
 b_{i,j}[0] &= a_{i,j}[5] \oplus a_{i,j}[6] \oplus a_{i,j}[7] \oplus a_{i+1,j}[5] \oplus a_{i+1,j}[7] \oplus a_{i+1,j}[0] \\
&\quad \oplus a_{i+2,j}[0] \oplus a_{i+2,j}[5] \oplus a_{i+2,j}[6] \oplus a_{i+3,j}[5] \oplus a_{i+3,j}[0] \\
 b_{i,j}[1] &= a_{i,j}[5] \oplus a_{i,j}[0] \oplus a_{i+1,j}[6] \oplus a_{i+1,j}[5] \oplus a_{i+1,j}[0] \oplus a_{i+1,j}[7] \\
&\quad \oplus a_{i+1,j}[1] \oplus a_{i+2,j}[1] \oplus a_{i+2,j}[7] \oplus a_{i+2,j}[5] \oplus a_{i+3,j}[6] \oplus a_{i+3,j}[5] \oplus a_{i+3,j}[1] \\
 b_{i,j}[2] &= a_{i,j}[6] \oplus a_{i,j}[0] \oplus a_{i,j}[1] \oplus a_{i+1,j}[7] \oplus a_{i+1,j}[6] \oplus a_{i+1,j}[1] \\
&\quad \oplus a_{i+1,j}[2] \oplus a_{i+2,j}[0] \oplus a_{i+2,j}[2] \oplus a_{i+2,j}[6] \oplus a_{i+3,j}[7] \oplus a_{i+3,j}[6] \oplus a_{i+3,j}[2] \\
 b_{i,j}[3] &= a_{i,j}[0] \oplus a_{i,j}[5] \oplus a_{i,j}[1] \oplus a_{i,j}[6] \oplus a_{i,j}[2] \oplus a_{i,j}[2] \oplus a_{i+1,j}[0] \oplus a_{i+1,j}[5] \oplus a_{i+1,j}[3] \oplus a_{i+2,j}[0] \oplus a_{i+2,j}[1] \oplus a_{i+2,j}[3] \oplus a_{i+3,j}[5] \oplus a_{i+3,j}[0] \oplus a_{i+3,j}[7] \oplus a_{i+3,j}[5] \oplus a_{i+3,j}[3] \\
 b_{i,j}[4] &= a_{i,j}[1] \oplus a_{i,j}[5] \oplus a_{i,j}[2] \oplus a_{i,j}[3] \oplus a_{i+1,j}[1] \oplus a_{i+1,j}[6] \oplus a_{i+1,j}[5] \oplus a_{i+1,j}[3] \oplus a_{i+2,j}[0] \oplus a_{i+2,j}[1] \oplus a_{i+2,j}[4] \oplus a_{i+2,j}[4] \oplus a_{i+3,j}[5] \oplus a_{i+3,j}[3] \oplus a_{i+3,j}[7] \oplus a_{i+3,j}[4] \\
 b_{i,j}[5] &= a_{i,j}[2] \oplus a_{i,j}[6] \oplus a_{i,j}[3] \oplus a_{i,j}[4] \oplus a_{i+1,j}[2] \oplus a_{i+1,j}[7] \oplus a_{i+1,j}[6] \oplus a_{i+1,j}[5] \oplus a_{i+2,j}[0] \oplus a_{i+2,j}[2] \oplus a_{i+2,j}[3] \oplus a_{i+2,j}[5] \oplus a_{i+3,j}[6] \oplus a_{i+3,j}[5] \oplus a_{i+3,j}[2] \oplus a_{i+3,j}[6] \oplus a_{i+3,j}[5] \oplus a_{i+3,j}[7] \oplus a_{i+3,j}[5] \\
 b_{i,j}[6] &= a_{i,j}[3] \oplus a_{i,j}[7] \oplus a_{i,j}[4] \oplus a_{i,j}[5] \oplus a_{i+1,j}[3] \oplus a_{i+1,j}[7] \oplus a_{i+1,j}[5] \oplus a_{i+1,j}[6] \oplus a_{i+2,j}[3] \oplus a_{i+2,j}[4] \oplus a_{i+2,j}[7] \oplus a_{i+2,j}[7] \oplus a_{i+3,j}[3] \oplus a_{i+3,j}[7] \oplus a_{i+3,j}[6] \oplus a_{i+3,j}[6] \\
 b_{i,j}[7] &= a_{i,j}[4] \oplus a_{i,j}[5] \oplus a_{i,j}[6] \oplus a_{i+1,j}[4] \oplus a_{i+1,j}[6] \oplus a_{i+1,j}[7] \oplus a_{i+2,j}[4] \oplus a_{i+2,j}[5] \oplus a_{i+2,j}[7] \oplus a_{i+3,j}[4] \oplus a_{i+3,j}[7] \oplus a_{i+3,j}[7] \\
\end{align*}
\]

Figure 3.5: InvMixColumns equations

3.3 Bitsliced AddRoundKey

AddRoundKey is its own inverse. Therefore, there is no difference between the AddRoundKey of encryption and decryption algorithm except that AddRoundKey has been separated out from ShiftBytes in the decryption algorithm.
3.4 Bitsliced InvSubBytes

InvSubBytes starts by subtracting (using XOR) the NIST-specified constant from the byte, followed by the inverse of the affine transformation. Using the result, the multiplicative inverse is calculated. The inverse of affine transformation and the basis change matrix, from standard basis to subfield basis, are combined to form matrix $MX^{-1}$, shown in Figure 3.6. After the subtraction of the constant, the byte is multiplied by $MX^{-1}$, the multiplicative inverse is calculated, and the result is changed from subfield basis to standard basis using matrix $X$, shown in Figure 3.7.

$$
(MX)^{-1} = \begin{pmatrix}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & 1
\end{pmatrix}
$$

**Figure 3.6: Inverse affine transform and standard to subfield basis change**

$$
\begin{pmatrix}
g_7 \\
g_6 \\
g_5 \\
g_4 \\
g_3 \\
g_2 \\
g_1 \\
g_0
\end{pmatrix} = \begin{pmatrix}
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{pmatrix} \begin{pmatrix}
b_7 \\
b_6 \\
b_5 \\
b_4 \\
b_3 \\
b_2 \\
b_1 \\
b_0
\end{pmatrix}
$$

**Figure 3.7: Subfield to standard basis change matrix**
3.5 Bitsliced Inverse Key Expansion Algorithm

Since, in this paper, the decryption algorithm used is the straightforward Inverse Cipher, the key expansion algorithm for the inverse cipher is the same as the key expansion algorithm for the encryption cipher. However, the optimization that was added to the key expansion algorithm by Käsper and Schwabe has been removed for the decryption algorithm. Because the order of XOR by constant in InvSubBytes has been changed for the decryption algorithm, the optimization is not feasible anymore.
Chapter 4
VALIDATION AND PERFORMANCE ANALYSIS

To verify the correctness of the decryption algorithm, the output of the bitsliced algorithm was compared against the output of the OpenSSL implementation of the AES decryption algorithm. The bitsliced decryption algorithm was plugged into the eSTREAM benchmarking tool. eSTREAM tool goes through many types of input for the algorithm and produces the output in a file. It also collects data about algorithm execution speed. The OpenSSL algorithm was plugged into the same tool, so it can execute the standard implementation of the algorithm for the same inputs as the bitsliced version. The OpenSSL test code is shown in Appendix B. The outputs produced by the OpenSSL implementation and the bitsliced implementation were the same.

The performance data for bitsliced encryption and bitsliced decryption, along with other implementation, is shown in Table 4.1. Shown in the Table 4.1 is cycles per byte it takes to process a stream of 128-bit blocks, 40 bytes packet, 576 byte packets, 1500 byte packets, key expansion setup, and IV setup. This performance data is collected on a Core2Duo CPU. Full system configuration is shown in Appendix C. Bitsliced encryption by Käsper and Schwabe takes 9.51 cycles per byte to process a 128-bit block. Bitsliced decryption algorithm described in this report takes 12.89 cycles per byte to process a 128-bit block. To decrypt the same block, an OpenSSL implementation takes 34.38 cycles per byte.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Stream</th>
<th>40 bytes</th>
<th>576 bytes</th>
<th>1500 bytes</th>
<th>Key setup</th>
<th>IV setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitsliced Encryption</td>
<td>9.51</td>
<td>34.98</td>
<td>11.01</td>
<td>9.98</td>
<td>1413.98</td>
<td>26.67</td>
</tr>
<tr>
<td>Bitsliced Decryption</td>
<td>12.89</td>
<td>45.34</td>
<td>14.79</td>
<td>13.42</td>
<td>1398.44</td>
<td>26.88</td>
</tr>
<tr>
<td>bernstein/amd64-2/1</td>
<td>13.76</td>
<td>20.28</td>
<td>13.99</td>
<td>13.88</td>
<td>173.8</td>
<td>31.84</td>
</tr>
<tr>
<td>bernstein/amd64-1/1</td>
<td>14.34</td>
<td>20.98</td>
<td>14.61</td>
<td>14.51</td>
<td>174.96</td>
<td>31.77</td>
</tr>
<tr>
<td>gladman/1</td>
<td>34.13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>OpenSSL Decryption</td>
<td>34.38</td>
<td>42.16</td>
<td>34.38</td>
<td>34.45</td>
<td>1348.78</td>
<td>35.33</td>
</tr>
<tr>
<td>bernstein/little-2/1</td>
<td>48.18</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>bernstein/little-4/1</td>
<td>50.54</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>bernstein/big-1/1</td>
<td>53.91</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>bernstein/little-1/1</td>
<td>54.61</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>hongjun/v1/1</td>
<td>57.58</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>hongjun/v0/1</td>
<td>58.78</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>bernstein/little-3/1</td>
<td>68.13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.1: Performance data for Core2Duo CPU

The bitsliced decryption algorithm is faster than the OpenSSL implementation; however, it is slower than the bitsliced encryption algorithm. Even though the bitsliced decryption is slower than bitsliced encryption algorithm, it is still faster than any of the AES-CTR benchmark encryption implementations available in eSTREAM package. This slowness, however, is expected because the InvSubBytes requires more complicated multiplication operations, which in turn requires more instructions to execute. The total number of instructions used for bitsliced encryption and bitsliced decryption operations are shown in Table 4.2.
As seen in Table 4.2, InvMixColumns requires 99 more instructions than the MixColumns operations used in encryption algorithm. The total number of instructions needed for 1 round of bitsliced encryption is 222. The total number of instructions needed for 1 round of bitsliced decryption is 335. Due to the increased amount of instructions needed for the algorithm, the decryption algorithm takes more cycles than the encryption algorithm to process one block of data.
Encryption is becoming widespread as internet communication is becoming widely accessible. Due to this importance, many encryption algorithms have been invented. AES is one of the widely used encryption and therefore many optimizations to speed up the encryption have been proposed. Käsper and Schwabe proposed a bitsliced implementation, which is not only the fastest AES implementation yet but also protects against cache-timing attacks by making the algorithm a constant time operation.

This paper detailed the decryption algorithm to go along with Käsper and Schwabe’s encryption implementation. The decryption algorithm, while faster than any encryption algorithm in the eSTREAM benchmark, was slower than the bitsliced encryption algorithm. This decrease in speed is expected due to the InvMixColumns operation being more complicated for decryption than the MixColumns operation for encryption. The bitsliced decryption algorithm is compared against OpenSSL to validate that the output of the decryption is correct. The performance testing showed that the bitsliced decryption algorithm is almost 3 times as fast as the standard OpenSSL AES decryption implementation.

Intel has announced that AES instructions will be built into their processors [5]. While this hardware implementation of AES is faster than the software implementation and will prevent against cache timing attacks, the widespread usage of these processors will some time, especially in the corporate market where adoption of newer technology is slow. Until the new hardware is in widespread use, the bitsliced implementation
proposed by Käsper and the bitsliced implementation described in this paper will be able
to provide the speed and security that is not available in standard AES implementations
such as OpenSSL.
APPENDIX A

Bitsliced Decryption Code

ifndef ADDROUNDKEY_S
ADDROUNDKEY_S:

# Add round key
.macro AddRoundKey x0, x1, x2, x3, x4, x5, x6, x7, i, k

    p xor (128*(\i))(\k), \x0
    p xor (128*(\i)+16)(\k), \x1
    p xor (128*(\i)+32)(\k), \x2
    p xor (128*(\i)+48)(\k), \x3
    p xor (128*(\i)+64)(\k), \x4
    p xor (128*(\i)+80)(\k), \x5
    p xor (128*(\i)+96)(\k), \x6
    p xor (128*(\i)+112)(\k), \x7

.endm

.endif

ifndef INVMIXCOLUMNS_S
INVMIXCOLUMNS_S:

.macro invMixColumns x0, x1, x2, x3, x4, x5, x6, x7, t0, t1, t2, t3, t4, t5, t6, t7

#load initial values
    movdqa \x5, \t0
    movdqa \x5, \t1
    movdqa \x6, \t2
    movdqa \x0, \t3
    movdqa \x1, \t4
    movdqa \x2, \t5
    movdqa \x3, \t6
    movdqa \x4, \t7

#xor rest of A_i values
    p xor \x6, \t0
    p xor \x7, \t0
    p xor \x0, \t1
    p xor \x0, \t2
    p xor \x1, \t2
pxor \x5, \t3
pxor \x1, \t3
pxor \x6, \t3
pxor \x2, \t3
pxor \x5, \t4
pxor \x2, \t4
pxor \x3, \t4
pxor \x6, \t5
pxor \x3, \t5
pxor \x4, \t5
pxor \x7, \t6
pxor \x4, \t6
pxor \x5, \t6
pxor \x5, \t7
pxor \x6, \t7

# x <<< 32: x becomes A_i+1
pshufd $0x93, \x0, \x0
pshufd $0x93, \x1, \x1
pshufd $0x93, \x2, \x2
pshufd $0x93, \x3, \x3
pshufd $0x93, \x4, \x4
pshufd $0x93, \x5, \x5
pshufd $0x93, \x6, \x6
pshufd $0x93, \x7, \x7

# xor with A_i+1
pxor \x5, \t0
pxor \x7, \t0
pxor \x0, \t0
pxor \x6, \t1
pxor \x5, \t1
pxor \x0, \t1
pxor \x7, \t1
pxor \x1, \t1
pxor \x7, \t2
pxor \x6, \t2
pxor \x1, \t2
pxor \x2, \t2
pxor \x0, \t3
pxor \x5, \t3
pxor \x2, \t3
pxor \x3, \t3
pxor \x1, \t4
# (x << 32) <<< 32 = x <<< 64. x becomes A_{i+2}

pshufd $0x93, \x0, \x0
pshufd $0x93, \x1, \x1
pshufd $0x93, \x2, \x2
pshufd $0x93, \x3, \x3
pshufd $0x93, \x4, \x4
pshufd $0x93, \x5, \x5
pshufd $0x93, \x6, \x6
pshufd $0x93, \x7, \x7

# xor with A_{i+2}
pxor \x0, \t0
pxor \x5, \t0
pxor \x6, \t0
pxor \x1, \t1
pxor \x7, \t1
pxor \x5, \t1
pxor \x0, \t2
pxor \x2, \t2
pxor \x6, \t2
pxor \x0, \t3
pxor \x1, \t3
pxor \x3, \t3
pxor \x5, \t3
pxor \x6, \t3
pxor \x7, \t3
pxor \x1, \t4
pxor \x2, \t4
pxor \x4, \t4
pxor \x5, \t4
pxor \x7, \t4
pxor \x2, \t5
pxor \x3, \t5
pxor \x5, \t5
pxor \x6, \t5
pxor \x3, \t6
pxor \x4, \t6
pxor \x6, \t6
pxor \x7, \t6
pxor \x4, \t7
pxor \x5, \t7
pxor \x7, \t7

# (x <<< 64) <<< 32 = x <<< 96. x becomes A_{i+3}
pshufd $0x93, \x0, \x0
pshufd $0x93, \x1, \x1
pshufd $0x93, \x2, \x2
pshufd $0x93, \x3, \x3
pshufd $0x93, \x4, \x4
pshufd $0x93, \x5, \x5
pshufd $0x93, \x6, \x6
pshufd $0x93, \x7, \x7

#xor with A_{i+3}
pxor \x5, \t0
pxor \x0, \t0
pxor \x6, \t1
pxor \x5, \t1
pxor \x1, \t1
pxor \x7, \t2
pxor \x6, \t2
pxor \x2, \t2
pxor \x0, \t3
pxor \x5, \t3
pxor \x7, \t3
pxor \x3, \t3
pxor \x1, \t4
pxor \x5, \t4
pxor \x6, \t4
pxor \x4, \t4
pxor \x2, \t5
pxor \x6, \t5
pxor \x7, \t5
pxor \x5, \t5
pxor \x3, \t6
pxor \x7, \t6
pxor \x6, \t6
pxor \x4, \t7
pxor \x7, \t7
.endm
.endif

.ifndef INVSHIFTROWS_S
INVSHIFTROWS_S:

# Inverse Shift Rows
.macro invShiftRows x0, x1, x2, x3, x4, x5, x6, x7, M
  pshufb \M, \x0
  pshufb \M, \x1
  pshufb \M, \x2
  pshufb \M, \x3
  pshufb \M, \x4
  pshufb \M, \x5
  pshufb \M, \x6
  pshufb \M, \x7
.endm
.endif

.ifndef INVSUBBYTES_S
INVSUBBYTES_S:

# Inverse sbox
# input in lsb   > [b0, b1, b2, b3, b4, b5, b6, b7] < msb
# output in lsb  > [b5, b3, b7, b6, b2, b0, b4, b1] < msb
.macro invSbox b0, b1, b2, b3, b4, b5, b6, b7, t0, t1, t2, t3, s0, s1, s2, s3
  #subtract
  pxor ONE, \b6
  pxor ONE, \b5
  pxor ONE, \b1
pxor ONE, \b0

invInBasisChange \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7
# output in lsb   > [b1, b3, b2, b7, b0, b4, b5, b6] < msb

Inv_GF256 \b1, \b3, \b2, \b7, \b0, \b4, \b5, \b6, \t0, \t1, \t2, \t3, \s0, \s1, \s2, \s3
# output lsb > [x4,x5,x6,x7,x0,x1,x2,x3] < msb

invOutBasisChange \b0, \b4, \b5, \b6, \b1, \b3, \b2, \b7
# output in lsb   > [x2, x5, x7, x3, x6, x0, x1, x4] < msb

.endm

# Inverse InBasisChange
# input in lsb   > [b0, b1, b2, b3, b4, b5, b6, b7] < msb
# output in lsb   > [b1, b3, b2, b7, b0, b4, b5, b6] < msb
.macro invInBasisChange b0, b1, b2, b3, b4, b5, b6, b7
        pxor \b5, \b2
        pxor \b7, \b2
        pxor \b6, \b0
        pxor \b6, \b4
        pxor \b0, \b3
        pxor \b4, \b3
        pxor \b4, \b7
        pxor \b7, \b6
        pxor \b0, \b1
        pxor \b5, \b1
        pxor \b4, \b1
        pxor \b7, \b1
        pxor \b6, \b1
        pxor \b1, \b5
        pxor \b3, \b0
        pxor \b5, \b0
        pxor \b7, \b0
        pxor \b6, \b0
.endm

# Inverse OutBasisChange
# input in lsb   > [b0, b1, b2, b3, b4, b5, b6, b7] < msb
# output in lsb   > [b2, b5, b7, b3, b6, b0, b1, b4] < msb
.macro invOutBasisChange b0, b1, b2, b3, b4, b5, b6, b7
        pxor \b1, \b4
.endm
.pxor \b1, \b5
.pxor \b1, \b6
.pxor \b7, \b1

.pxor \b3, \b1
.pxor \b0, \b1
.pxor \b5, \b1
.pxor \b6, \b1

.pxor \b2, \b3
.pxor \b4, \b3
.pxor \b5, \b3
.pxor \b6, \b3

.pxor \b3, \b0
.pxor \b4, \b0
.pxor \b7, \b0
.pxor \b1, \b7

.pxor \b0, \b7
.pxor \b4, \b7
.pxor \b5, \b7
.pxor \b2, \b7

.endm

.endif

.ifndef INVAES128_S
INVAES128_S:

# AES-128 Decryption
# b0-b7 - input ciphertext
# t0-t7 - output plaintext
# k - key
.macro invAes128 b0, b1, b2, b3, b4, b5, b6, b7, t0, t1, t2, t3, t4, t5, t6, t7, k

# bitslice input. t0 is temporary register
bitslice \b7, \b6, \b5, \b4, \b3, \b2, \b1, \b0, \t0

# initial key addition
AddRoundKey \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, 10, \k

# begin AES rounds
invAesRound 9, \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7, \k
invAesRound 8, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7, \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \k
invAesRound 7, \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7, \k

.invaesround 6, \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7, \k

.invaesround 5, \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7, \k

.invaesround 4, \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7, \k

.invaesround 3, \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7, \k

.invaesround 2, \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7, \k

.invaesround 1, \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7, \k

.invaesround 0, \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7, \k

.endm

.endif
invAesRound 6, \(t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, k\)
invAesRound 5, \(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, k\)
invAesRound 4, \(t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, k\)
invAesRound 3, \(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, k\)
invAesRound 2, \(t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, k\)
invAesRound 1, \(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, k\)
invLastRound \(t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, k\)
\# output in lsb > \([t_5, t_3, t_7, t_6, t_2, t_0, t_4, t_1]\) < msb
\# call bitslice again to invert bitslicing
bitslice \(t_1, t_4, t_0, t_2, t_6, t_7, t_3, t_5, b_0\)

\# invAesRound
.macro invAesRound i, b0, b1, b2, b3, b4, b5, b6, b7, t0, t1, t2, t3, t4, t5, t6, t7, k
invShiftRows \(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, ISR\)
invSbox \(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7\)
\# output in lsb > \([b_5, b_3, b_7, b_6, b_2, b_0, b_4, b_1]\) < msb
AddRoundKey \(b_5, b_3, b_7, b_6, b_2, b_0, b_4, b_1, i, k\)
invMixColumns \(b_5, b_3, b_7, b_6, b_2, b_0, b_4, b_1, t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7\)
\# output in lsb > \([t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7]\) < msb
.endm

\# Last AES Round
.macro invLastRound b0, b1, b2, b3, b4, b5, b6, b7, t0, t1, t2, t3, t4, t5, t6, t7, k
invShiftRows \(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, ISR\)
invSbox \(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7\)
# output in lsb  > [b5, b3, b7, b6, b2, b0, b4, b1] < msb

AddRoundKey \b5, \b3, \b7, \b6, \b2, \b0, \b4, \b1, 0, \k

.endm

.endif


#int action in %edi
#ECRYPT_ctx* ctx in %rsi
#const u8* input in %rdx
#u8* output in %rcx
#u32 msglen in %r8d

.globl ECRYPT_process_bytes
ECRYPT_process_bytes:
cmpl $0, %r8d
jne .START
ret

.START:
# bitsliced key
lea (\rsi), %rax

.pushq %rbx
.pushq %rbp
.pushq %r12
.pushq %r13
.pushq %r15

#msglen
movl %r8d, %r12d
#input
movq %rdx, %rbx
#output
movq %rcx, %rbp

.ENC_BLOCK:
    movdqa 1408(%rsi), %xmm0
    movdqa %xmm0, %xmm1
    movdqa %xmm0, %xmm2
    movdqa %xmm1, %xmm3
    movdqa %xmm0, %xmm4
    movdqa %xmm2, %xmm5
    movdqa %xmm1, %xmm6
    movdqa %xmm3, %xmm7
    paddd CTRINC1, %xmm1
    paddd CTRINC2, %xmm2
paddd CTRINC3, %xmm3
paddd CTRINC4, %xmm4
paddd CTRINC5, %xmm5
paddd CTRINC6, %xmm6
paddd CTRINC7, %xmm7

pshufb M0, %xmm0
pshufb M0, %xmm1
pshufb M0, %xmm2
pshufb M0, %xmm3
pshufb M0, %xmm4
pshufb M0, %xmm5
pshufb M0, %xmm6
pshufb M0, %xmm7

invAes128 %xmm0, %xmm1, %xmm2, %xmm3, %xmm4, %xmm5, %xmm6, %xmm7,
%xmm8, %xmm9, %xmm10, %xmm11, %xmm12, %xmm13, %xmm14, %xmm15, %rax

pshufb M0, %xmm13
pshufb M0, %xmm11
pshufb M0, %xmm15
pshufb M0, %xmm14
pshufb M0, %xmm10
pshufb M0, %xmm8
pshufb M0, %xmm12
pshufb M0, %xmm9

# output in first block > [xmm13, xmm11, xmm15, xmm14, xmm10, xmm8,
xmm12, xmm9] < last block

cmpl $128, %r12d
jb .PARTIAL
je .FULL

addl $8, 1408(%rsi)
pxor (%rbx), %xmm13
pxor 16(%rbx), %xmm11
pxor 32(%rbx), %xmm15
pxor 48(%rbx), %xmm14
pxor 64(%rbx), %xmm10
pxor 80(%rbx), %xmm8
pxor 96(%rbx), %xmm12
pxor 112(%rbx), %xmm9

movdqa %xmm13, (%rbp)
movdqa %xmm11, 16(%rbp)
movdqa %xmm15, 32(%rbp)
movdqa %xmm14, 48(%rbp)
movdqa %xmm10, 64(%rbp)
movdqa %xmm8, 80(%rbp)
movdqa %xmm12, 96(%rbp)
movdqa %xmm9, 112(%rbp)
subl $128, %r12d
addq $128, %rbx
deq $128, %rbp
jmp .ENC_BLOCK

.PARTIAL:
# add partial bytes #
movl %r12d, %r13d
shr $4, %r12d
addl %r12d, 1408(%rsi)
movq %rsp, %r15
subq $128, %rsp
and $0xFFFFFFFFFFFFFF00, %rsp
movdqa %xmm13, (%rsp)
movdqa %xmm11, 16(%rsp)
movdqa %xmm15, 32(%rsp)
movdqa %xmm14, 48(%rsp)
movdqa %xmm10, 64(%rsp)
movdqa %xmm8, 80(%rsp)
movdqa %xmm12, 96(%rsp)
movdqa %xmm9, 112(%rsp)

.BYTES:

movb (%rbx), %al
xorb (%rsp), %al
movb %al, (%rbp)
addq $1, %rbx
addq $1, %rbp
addq $1, %rsp
subl $1, %r13d
cmp $0, %r13d
jne .BYTES
movq %r15, %rsp
jmp .END

.FULL:

addl $8, 1408(%rsi)
pxor (%rbx), %xmm13
pxor 16(%rbx), %xmm11
pxor 32(%rbx), %xmm15
pxor 48(%rbx), %xmm14
pxor 64(%rbx), %xmm10
pxor 80(%rbx), %xmm8
pxor 96(%rbx), %xmm12
pxor 112(%rbx), %xmm9

movdqa %xmm13, (%rbp)
movdqa %xmm11, 16(%rbp)
movdqa %xmm15, 32(%rbp)
movdqa %xmm14, 48(%rbp)
movdqa %xmm10, 64(%rbp)
movdqa %xmm8, 80(%rbp)
movdqa %xmm12, 96(%rbp)
movdqa %xmm9, 112(%rbp)
.END:
popq   %r15
popq   %r13
popq   %r12
popq   %rbp
popq   %rbx
ret

.data
.globl  RCON
.globl  ROTB
.globl  EXPB0
.globl  ONE
.globl  BS0
.globl  BS1
.globl  BS2
.globl  CTRINC1
.globl  CTRINC2
.globl  CTRINC3
.globl  CTRINC4
.globl  CTRINC5
.globl  CTRINC6
.globl  CTRINC7
.globl  M0
.globl  SRM0
.globl  SR
.globl  ISR

.p2align 6
#.align 16
#.section .rodata
RCON: .int 0x00000000, 0x00000000, 0x00000000, 0xffffffff
ROTB: .int 0x0c000000, 0x00000000, 0x04000000, 0x08000000
EXPB0: .int 0x03030303, 0x07070707, 0x0b0b0b0b, 0x0f0f0f0f
CTRINC1: .int 0x00000001, 0x00000000, 0x00000000, 0x00000000
CTRINC2: .int 0x00000002, 0x00000000, 0x00000000, 0x00000000
CTRINC3: .int 0x00000003, 0x00000000, 0x00000000, 0x00000000
CTRINC4: .int 0x00000004, 0x00000000, 0x00000000, 0x00000000
CTRINC5: .int 0x00000005, 0x00000000, 0x00000000, 0x00000000
CTRINC6: .int 0x00000006, 0x00000000, 0x00000000, 0x00000000
CTRINC7: .int 0x00000007, 0x00000000, 0x00000000, 0x00000000
BS0:    .quad 0x5555555555555555, 0x5555555555555555
BS1:    .quad 0x3333333333333333, 0x3333333333333333
BS2:    .quad 0x0f0f0f0f0f0f0f0f, 0x0f0f0f0f0f0f0f0f
ONE:    .quad 0xfffffffffffffff, 0xfffffffffffffff
M0:    .quad 0x02060a0e03070b0f, 0x0004080c0105090d
ISRMO: .quad 0x01040b0e0205080f, 0x0306090c00070a0d
SRM0: .quad 0x0304090e00050a0f, 0x01060b0c0207080d
SR: .quad 0x0504070600030201, 0x0f0e0d0c0a09080b
ISR: .quad 0x0504070602010003, 0x0f0e0d0c080b0a09

.ifndef COMMON_S
COMMON_S:

    # sbox
    # input in lsb > [b0, b1, b2, b3, b4, b5, b6, b7] < msb
    # output in lsb > [b0, b1, b4, b6, b3, b7, b2, b5] < msb

    .macro sbox b0, b1, b2, b3, b4, b5, b6, b7, t0, t1, t2, t3, s0, s1, s2, s3
        InBasisChange \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7
        Inv_GF256 \b6, \b5, \b0, \b3, \b7, \b1, \b4, \b2, \t0, \t1, \t2, \t3,
        \s0, \s1, \s2, \s3
        OutBasisChange \b7, \b1, \b4, \b2, \b6, \b5, \b0, \b3
    .endm

    # InBasisChange
    # input in lsb > [b0, b1, b2, b3, b4, b5, b6, b7] < msb
    # output in lsb > [b6, b5, b0, b3, b7, b1, b4, b2] < msb

    .macro InBasisChange b0, b1, b2, b3, b4, b5, b6, b7
        pxor \b6, \b5
        pxor \b1, \b2
        pxor \b0, \b5
        pxor \b2, \b6
        pxor \b0, \b3
        pxor \b3, \b6
        pxor \b7, \b3
        pxor \b4, \b3
        pxor \b5, \b7
        pxor \b1, \b3
        pxor \b5, \b4
        pxor \b7, \b2
        pxor \b5, \b1
    .endm

    # OutBasisChange
    # input in lsb > [b0, b1, b2, b3, b4, b5, b6, b7] < msb

    .macro OutBasisChange b0, b1, b2, b3, b4, b5, b6, b7
        pxor \b7, \b6
        pxor \b5, \b4
        pxor \b7, \b2
        pxor \b5, \b1
    .endm
# output in lsb > [b6, b1, b2, b4, b7, b0, b3, b5] < msb

.macro OutBasisChange b0, b1, b2, b3, b4, b5, b6, b7
    pxor \b6, \b0
    pxor \b4, \b1
    pxor \b0, \b2
    pxor \b6, \b4
    pxor \b1, \b6
    pxor \b5, \b1
    pxor \b3, \b5
    pxor \b5, \b2
    pxor \b7, \b3
    pxor \b5, \b7
    pxor \b7, \b4
    # can modify subkeys instead, but won't for simplicity
    # it'll affect performance negatively
    pxor ONE, \b3
    pxor ONE, \b0
    pxor ONE, \b1
    pxor ONE, \b6
.endm

#;***************************************************************
#;* Mul_GF4: Input x0-x1,y0-y1 Output x0-x1 Temp t0 (8) *
#;***************************************************************

# Mul_GF4

.macro Mul_GF4 x0, x1, y0, y1, t0
    movdqa \y0, \t0
    pxor \y1, \t0
    pand \x0, \t0
    pxor \x1, \x0
    pand \y1, \x0
    pand \y0, \x1
    pxor \x1, \x0
    pxor \t0, \x1
.endm

# multiply and scale by N

.macro Mul_GF4_N x0, x1, y0, y1, t0
    movdqa \y0, \t0
    pxor \y1, \t0
    pand \x0, \t0
    pxor \x1, \x0
    pand \y1, \x0
    pand \y0, \x1
    pxor \x1, \x0
    pxor \t0, \x1
.endm
# multiply with common factor y

.macro Mul_GF4_2 x0, x1, x2, x3, y0, y1, t0, t1

movdqa \y0, \t0
pxor \y1, \t0
movdqa \t0, \t1
pand \x0, \t0
pand \x2, \t1
pxor \x1, \x0
pxor \x3, \x2
pand \y1, \x0
pand \y1, \x2
pand \y0, \x1
pand \y0, \x3
pxor \x1, \x0
pxor \x3, \x2
pxor \t0, \x1
pxor \t1, \x3
.endm

#;**********************************************
#;* Inv_GF4: Input x0,x1 Output x1 ,x0 (0)     *
#;**********************************************
# inversion is swapping

#;******************************************************************
#;* Mul_GF16: Input x0-x3,y0-y3 Output x0-x3 Temp t0-t3 (34)       *
#;******************************************************************

.macro Mul_GF16 x0, x1, x2, x3, y0, y1, y2, y3, t0, t1, t2, t3

movdqa \x0, \t0
movdqa \x1, \t1
Mul_GF4 \x0, \x1, \y0, \y1, \t2
pxor \x2, \t0
pxor \x3, \t1
pxor \y2, \y0
pxor \y3, \y1
Mul_GF4_N \t0, \t1, \y0, \y1, \t2
Mul_GF4 \x2, \x3, \y2, \y3, \t3
pxor \t0, \x0
pxor \t0, \x2
pxor \t1, \x1
pxor \t1, \x3
.endm

# multiply with common factors

.macro Mul_GF16_2 x0, x1, x2, x3, x4, x5, x6, x7, y0, y1, y2, y3, t0, t1, t2, t3
movdqa \x0, \t0
movdqa \x1, \t1
Mul_GF4 \x0, \x1, \y0, \y1, \t2
pxor \x2, \t0
pxor \x3, \t1
pxor \y2, \y0
pxor \y3, \y1
Mul_GF4_N \t0, \t1, \y0, \y1, \t3
Mul_GF4-\x2, \x3, \y2, \y3, \t2
pxor \t0, \x0
pxor \t0, \x2
pxor \t1, \x1
pxor \t1, \x3
movdqa \x4, \t0
movdqa \x5, \t1
pxor \x6, \t0
pxor \x7, \t1
Mul_GF4_N \t0, \t1, \y0, \y1, \t3
Mul_GF4-\x6, \x7, \y2, \y3, \t2
pxor \y2, \y0
pxor \y3, \y1
Mul_GF4-\x4, \x5, \y0, \y1, \t3
pxor \t0, \x4
pxor \t0, \x6
pxor \t1, \x5
pxor \t1, \x7
.endm

#;******************************************************
#;* Inv_GF16: Input x0-x3 Output x0-x3 Temp t0-t3 (26) *
#;******************************************************

# Inv_GF16
.macro Inv_GF16 x0, x1, x2, x3, t0, t1, t2, t3
    movdqa \x1, \t0
    movdqa \x0, \t1
    pand \x3, \t0
    por \x2, \t1
    movdqa \x1, \t2
    movdqa \x0, \t3
    por \x2, \t2
    por \x3, \t3
    pxor \t3, \t2
    pxor \t2, \t0
    pxor \t2, \t1
    # inversion for (t0, t1) is swapping
    Mul_GF4_2 \x0, \x1, \x2, \x3, \t1, \t0, \t2, \t3
.endm

# ;********************************************************************
# ;* Inv_GF256: Input x0-x7 Output x0-x7 Temp t0-t3,s0-s3 (144)       *
# ;********************************************************************
#
# Inv_GF256

.macros Inv_GF256 x0,x1,x2,x3,x4,x5,x6,x7,t0,t1,t2,t3,s0,s1,s2,s3
    # direct optimizations from hardware
    movdqa \x4, \t3
    movdqa \x5, \t2
    movdqa \x1, \t1
    movdqa \x7, \s1
    movdqa \x0, \s0
    pxor \x6, \t3
    pxor \x7, \t2
    pxor \x3, \t1
    pxor \x6, \s1
    pxor \x2, \s0
    movdqa \t3, \s2
    movdqa \t2, \t0
    movdqa \t3, \s3
    por \t1, \t2
    por \s0, \t3
    pxor \t0, \s3
.endmacs
pand \s0, \s2
pand \t1, \t0
pxor \t1, \s0
pand \s0, \s3
movdqa \x3, \s0
pxor \x2, \s0
pand \s0, \s1
pxor \s1, \t3
pxor \s1, \t2
movdqa \x4, \s1
pxor \x5, \s1
movdqa \x1, \s0
movdqa \s1, \t1
pxor \x0, \s0
por \s0, \t1
pand \s0, \t1
pxor \s1, \t0
pxor \s3, \t3
pxor \s2, \t2
pxor \s3, \t1
pxor \s2, \t0
pxor \s2, \t1
movdqa \x7, \s0
movdqa \x6, \s1
movdqa \x5, \s2
movdqa \x4, \s3
pand \x3, \s0
pand \x2, \s1
pand \x1, \s2
por \x0, \s3
pxor \s0, \t3
pxor \s1, \t2
pxor \s2, \t1
pxor \s3, \t0

#Inv_GF16 \t0, \t1, \t2, \t3, \s0, \s1, \s2, \s3

# new smaller inversion

movdqa \t3, \s0
pxor \t2, \s0
pand \t1, \t3
movdqa \t0, \s2
pxor \t3, \s2
movdqa \s0, \s3
pand \s2, \s3
pxor \t2, \s3
movdqa \t1, \s1
pxor \t0, \s1
pxor \t2, \t3
pand \t3, \s1
pxor \t0, \s1
pxor \s1, \t1
movdqa \s2, \t2
pxor \s1, \t2
pand \t0, \t2
pxor \t2, \t1
pxor \t2, \s2
pand \s3, \s2
pxor \s0, \s2

# output in s3, s2, s1, t1

# Mul_GF16_2 \x0, \x1, \x2, \x3, \x4, \x5, \x6, \x7, \t2, \t3, \t0, \t1, \s0, \s1, \s2, \s3
Mul_GF16_2 \x0, \x1, \x2, \x3, \x4, \x5, \x6, \x7, \s3, \s2, \s1, \t1, \s0, \t0, \t2, \t3

### output msb > [x3,x2,x1,x0,x7,x6,x5,x4] < lsb
.endm

# AES linear components

.macros shiftrows x0, x1, x2, x3, x4, x5, x6, x7, i, M, k
pxor (128*(i-1))(\k), \x0
pshufb M, \x0
pxor (128*(i-1)+16)(\k), \x1
pshufb M, \x1
pxor (128*(i-1)+32)(\k), \x2
pshufb M, \x2
pxor (128*(i-1)+48)(\k), \x3
pshufb M, \x3
pxor (128*(i-1)+64)(\k), \x4
pshufb M, \x4
pxor (128*(i-1)+80)(\k), \x5
pshufb \M, \x5
pxor (128*(i-1)+96)(\k), \x6
pshufb \M, \x6
pxor (128*(i-1)+112)(\k),\x7
pshufb \M, \x7
.endm

.macro mixcolumns x0, x1, x2, x3, x4, x5, x6, x7, t0, t1, t2, t3, t4, t5, t6, t7
pshufd $0x93, \x0, \t0  # x0 <<< 32
pshufd $0x93, \x1, \t1
pshufd $0x93, \x2, \t2
pshufd $0x93, \x3, \t3
pshufd $0x93, \x4, \t4
pshufd $0x93, \x5, \t5
pshufd $0x93, \x6, \t6
pshufd $0x93, \x7, \t7
pxor \t0, \x0  # x0 ^ (x0 <<< 32)
pxor \t1, \x1
pxor \t2, \x2
pxor \t3, \x3
pxor \t4, \x4
pxor \t5, \x5
pxor \t6, \x6
pxor \t7, \x7
pxor \x7, \t0
pxor \x0, \t1
pxor \x1, \t2
pxor \x7, \t1
pxor \x2, \t3
pxor \x3, \t4
pxor \x4, \t5
pxor \x7, \t3
pxor \x5, \t6
pxor \x6, \t7
pxor \x7, \t4
pshufd $0x4E, \x0, \x0  # (x0 ^ (x0 <<< 32)) <<< 64)
pshufd $0x4E, \x1, \x1
pshufd $0x4E, \x2, \x2
pshufd $0x4E, \x3, \x3
pshufd $0x4E, \x4, \x4
pshufd $0x4E, \x5, \x5
pshufd $0x4E, \x6, \x6
pshufd $0x4E, \x7, \x7
pxor \x0, \t0
pxor \xl, \t1
pxor \x2, \t2
pxor \x3, \t3
pxor \x4, \t4
pxor \x5, \t5
pxor \x6, \t6
pxor \x7, \t7

.endm

.macros swapmove a, b, n, m, t
movdqa \b, \t
psrlq \n, \t
pxor \a, \t
pand \m, \t
pxor \t, \a
psllq \n, \t
pxor \t, \b

.endm

.macros bitslice x0, x1, x2, x3, x4, x5, x6, x7, t
swapmove \x0, \x1, $1, BS0, \t
swapmove \x2, \x3, $1, BS0, \t
swapmove \x4, \x5, $1, BS0, \t
swapmove \x6, \x7, $1, BS0, \t

swapmove \x0, \x2, $2, BS1, \t
swapmove \x1, \x3, $2, BS1, \t
swapmove \x4, \x6, $2, BS1, \t
swapmove \x5, \x7, $2, BS1, \t

swapmove \x0, \x4, $4, BS2, \t
swapmove \x1, \x5, $4, BS2, \t
swapmove \x2, \x6, $4, BS2, \t
swapmove \x3, \x7, $4, BS2, \t

.endm

.macros bitslicekey0 key, bskey
movdqa (\key), %xmm0
pshufb M0, %xmm0
movdqa %xmm0, %xmm1
movdqa %xmm0, %xmm2
movdqa %xmm0, %xmm3
movdqa %xmm0, %xmm4
movdqa %xmm0, %xmm5
movdqa %xmm0, %xmm6
movdqa %xmm0, %xmm7

.endm
bitslice %xmm7, %xmm6, %xmm5, %xmm4, %xmm3, %xmm2, %xmm1, %xmm0, %xmm8

movdqa %xmm0,    (\bskey)
movdqa %xmm1,  16(\bskey)
movdqa %xmm2,  32(\bskey)
movdqa %xmm3,  48(\bskey)
movdqa %xmm4,  64(\bskey)
movdqa %xmm5,  80(\bskey)
movdqa %xmm6,  96(\bskey)
movdqa %xmm7, 112(\bskey)
.endm

.macro xorkeys b0, b1, b2, b3, b4, b5, b6, b7, t0, t1, t2, t3, t4, t5, t6, t7
pxor \t0, \b0
pxor \t1, \b1
pxor \t2, \b4
pxor \t3, \b6
pxor \t4, \b3
pxor \t5, \b7
pxor \t6, \b2
pxor \t7, \b5
psrld $8,\t0
psrld $8,\t1
psrld $8,\t2
psrld $8,\t3
psrld $8,\t4
psrld $8,\t5
psrld $8,\t6
psrld $8,\t7
pxor \t0, \b0
pxor \t1, \b1
pxor \t2, \b4
pxor \t3, \b6
pxor \t4, \b3
pxor \t5, \b7
pxor \t6, \b2
pxor \t7, \b5
psrld $8,\t0
psrld $8,\t1
psrld $8,\t2
psrld $8,\t3
psrld $8,\t4
psrld $8,\t5
psrld $8,\t6
psrld $8,\t7

.endm
pxor \( t0, \ b0 \)
pxor \( t1, \ b1 \)
pxor \( t2, \ b4 \)
pxor \( t3, \ b6 \)
pxor \( t4, \ b3 \)
pxor \( t5, \ b7 \)
pxor \( t6, \ b2 \)
pxor \( t7, \ b5 \)

psrl $8, \( t0 \)
psrl $8, \( t1 \)
psrl $8, \( t2 \)
psrl $8, \( t3 \)
psrl $8, \( t4 \)
psrl $8, \( t5 \)
psrl $8, \( t6 \)
psrl $8, \( t7 \)

pxor \( t0, \ b0 \)
pxor \( t1, \ b1 \)
pxor \( t2, \ b4 \)
pxor \( t3, \ b6 \)
pxor \( t4, \ b3 \)
pxor \( t5, \ b7 \)
pxor \( t6, \ b2 \)
pxor \( t7, \ b5 \)

.endm

.macro keyexp1 \ b0, \ b1, \ b2, \ b3, \ b4, \ b5, \ b6, \ b7, \ t0, \ t1, \ t2, \ t3, \ t4, \ t5, \ t6, \ t7, \ bskey
pshufb ROTB, \ b0
pshufb ROTB, \ b1
pshufb ROTB, \ b2
pshufb ROTB, \ b3
pshufb ROTB, \ b4
pshufb ROTB, \ b5
pshufb ROTB, \ b6
pshufb ROTB, \ b7
sbox \ b0, \ b1, \ b2, \ b3, \ b4, \ b5, \ b6, \ b7, \ t0, \ t1, \ t2, \ t3, \ t4, \ t5, \ t6, \ t7
pxor RCON, \ b0
pshufb EXPB0, \ b0
pshufb EXPB0, \ b1
pshufb EXPB0, \ b4
pshufb EXPB0, \ b6
pshufb EXPB0, \b3
pshufb EXPB0, \b7
pshufb EXPB0, \b2
pshufb EXPB0, \b5

movdqa  0(\bskey), \t0
movdqa  16(\bskey), \t1
movdqa  32(\bskey), \t2
movdqa  48(\bskey), \t3
movdqa  64(\bskey), \t4
movdqa  80(\bskey), \t5
movdqa  96(\bskey), \t6
movdqa 112(\bskey), \t7

xorkeys \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7

movdqa \b0,128(\bskey)
movdqa \b1,144(\bskey)
movdqa \b4,160(\bskey)
movdqa \b6,176(\bskey)
movdqa \b3,192(\bskey)
movdqa \b7,208(\bskey)
movdqa \b2,224(\bskey)
movdqa \b5,240(\bskey)

.endm

.mcro keyexp9 b0, b1, b2, b3, b4, b5, b6, b7, t0, t1, t2, t3, t4, t5, t6, t7, bskey
#pxor ONE, \b0
#pxor ONE, \b1
#pxor ONE, \b5
#pxor ONE, \b6

pshufb ROTB, \b0
pshufb ROTB, \b1
pshufb ROTB, \b2
pshufb ROTB, \b3
pshufb ROTB, \b4
pshufb ROTB, \b5
pshufb ROTB, \b6
pshufb ROTB, \b7

sbox \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7
pxor RCON, \b0
pxor RCON, \b1
pxor RCON, \b6
pxor RCON, \b3
pshufb EXPB0, \b0
pshufb EXPB0, \b1
pshufb EXPB0, \b4
pshufb EXPB0, \b6
pshufb EXPB0, \b3
pshufb EXPB0, \b7
pshufb EXPB0, \b2
pshufb EXPB0, \b5

movdqa 1024(\bskey), \t0
movdqa 1040(\bskey), \t1
movdqa 1056(\bskey), \t2
movdqa 1072(\bskey), \t3
movdqa 1088(\bskey), \t4
movdqa 1104(\bskey), \t5
movdqa 1120(\bskey), \t6
movdqa 1136(\bskey), \t7

xorkeys \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7

movdqa \b0,1152(\bskey)
movdqa \b1,1168(\bskey)
movdqa \b4,1184(\bskey)
movdqa \b6,1200(\bskey)
movdqa \b3,1216(\bskey)
movdqa \b7,1232(\bskey)
movdqa \b2,1248(\bskey)
movdqa \b5,1264(\bskey)

.endm

.macro keyexp10 b0, b1, b2, b3, b4, b5, b6, b7, t0, t1, t2, t3, t4, t5, t6, t7, bskey
#pxor ONE, \b0
#pxor ONE, \b1
#pxor ONE, \b5
#pxor ONE, \b6

pshufb ROTB, \b0
pshufb ROTB, \b1
pshufb ROTB, \b2
pshufb ROTB, \b3
pshufb ROTB, \b4
pshufb ROTB, \b5
pshufb ROTB, \b6
pshufb ROTB, \b7

sbox \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7
pxor RCON, \b1
pxor RCON, \b4
pxor RCON, \b7
pxor RCON, \b3

pshufb EXPB0, \b0
pshufb EXPB0, \b1
pshufb EXPB0, \b4
pshufb EXPB0, \b6
pshufb EXPB0, \b3
pshufb EXPB0, \b7
pshufb EXPB0, \b2
pshufb EXPB0, \b5

movdqa 1152(\bskey), \t0
movdqa 1168(\bskey), \t1
movdqa 1184(\bskey), \t2
movdqa 1200(\bskey), \t3
movdqa 1216(\bskey), \t4
movdqa 1232(\bskey), \t5
movdqa 1248(\bskey), \t6
movdqa 1264(\bskey), \t7

xorkeys \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7

movdqa \b0,1280(\bskey)
movdqa \b1,1296(\bskey)
movdqa \b4,1312(\bskey)
movdqa \b6,1328(\bskey)
movdqa \b3,1344(\bskey)
movdqa \b7,1360(\bskey)
movdqa \b2,1376(\bskey)
movdqa \b5,1392(\bskey)

.endm

.macro keyexp b0, b1, b2, b3, b4, b5, b6, b7, t0, t1, t2, t3, t4, t5, t6, t7, rconbit, round, bskey

pshufb ROTB, \b0
pshufb ROTB, \b1
pshufb ROTB, \b2
pshufb ROTB, \b3
pshufb ROTB, \b4
pshufb ROTB, \b5
pshufb ROTB, \b6
pshufb ROTB, \b7

sbox \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7
pxor RCON, \rconbit

pushf EXPBO, \b0
pushf EXPBO, \b1
pushf EXPBO, \b4
pushf EXPBO, \b6
pushf EXPBO, \b3
pushf EXPBO, \b7
pushf EXPBO, \b2
pushf EXPBO, \b5

movdqa ((\round-1)*128 + 0)(\bskey), \t0
movdqa ((\round-1)*128 + 16)(\bskey), \t1
movdqa ((\round-1)*128 + 32)(\bskey), \t2
movdqa ((\round-1)*128 + 48)(\bskey), \t3
movdqa ((\round-1)*128 + 64)(\bskey), \t4
movdqa ((\round-1)*128 + 80)(\bskey), \t5
movdqa ((\round-1)*128 + 96)(\bskey), \t6
movdqa ((\round-1)*128 + 112)(\bskey), \t7

xorkeys \b0, \b1, \b2, \b3, \b4, \b5, \b6, \b7, \t0, \t1, \t2, \t3, \t4, \t5, \t6, \t7

movdqa \b0, (\round*128 + 0)(\bskey)
movdqa \b1, (\round*128 + 16)(\bskey)
movdqa \b4, (\round*128 + 32)(\bskey)
movdqa \b6, (\round*128 + 48)(\bskey)
movdqa \b3, (\round*128 + 64)(\bskey)
movdqa \b7, (\round*128 + 80)(\bskey)
movdqa \b2, (\round*128 + 96)(\bskey)
movdqa \b5, (\round*128 + 112)(\bskey)

.endm
APPENDIX B

OpenSSL Decryption Code

/* encrypt-sync.h */

/*
 * Header file for synchronous stream ciphers without authentication
 * mechanism.
*
 * *** Please only edit parts marked with ":[edit]". ***
 */

#ifndef ECRYPT_SYNC
#define ECRYPT_SYNC

#include "encrypt-portable.h"
#include <openssl/evp.h>
#include <stdio.h>

/* ------------------------------- */

/* Cipher parameters */

/*
 * The name of your cipher.
 */
#define ECRYPT_NAME "AES-CTR" /* [edit] */
#define ECRYPT_PROFILE "bench"

/*
 * Specify which key and IV sizes are supported by your cipher. A user
 * should be able to enumerate the supported sizes by running the
 * following code:
 *
 * for (i = 0; ECRYPT_KEYSIZE(i) <= ECRYPT_MAXKEYSIZE; ++i)
 * {
 *   keysize = ECRYPT_KEYSIZE(i);
 *   ...
 * }
 *
 * All sizes are in bits.
 */
#define ECRYPT_MAXKEYSIZE 128 /* [edit] */
#define ECRYPT_KEYSIZE(i) (128 + (i)*128) /* [edit] */
#define ECRYPT_MAXIVSIZE 128 /* [edit] */
#define ECRYPT_IVSIZE(i) (128 + (i)*32) /* [edit] */
/* Data structures */

/*
 * ECRYPT_ctx is the structure containing the representation of the
 * internal state of your cipher.
 */

#define Nr 10
#define Nk 4
#define Nb 4
#define ROUND_KEY_COUNT ((Nr + 1) * Nb)

typedef struct
{
    EVP_CIPHER_CTX aesctx;
    __int128_t ctr;
} __attribute__ ((aligned (16))) ECRYPT_ctx;

/* Mandatory functions */

/*
 * Key and message independent initialization. This function will be
 * called once when the program starts (e.g., to build expanded S-box
 * tables).
 */
void ECRYPT_init(void);

/*
 * Key setup. It is the user's responsibility to select the values of
 * keysize and ivsize from the set of supported values specified
 * above.
 */
void ECRYPT_keysetup(
    ECRYPT_ctx* ctx,
    const u8* key,
    u32 keysize,        /* Key size in bits. */
    u32 ivsize);        /* IV size in bits. */

/*
 * IV setup. After having called ECRYPT_keysetup(), the user is
 * allowed to call ECRYPT_ivsetup() different times in order to
 * encrypt/decrypt different messages with the same key but different
 * IVs.
 */
void ECRYPT_ivsetup(
    ECRYPT_ctx* ctx,
    const u8* iv);
/* Encryption/decryption of arbitrary length messages. 
*
* For efficiency reasons, the API provides two types of 
* encrypt/decrypt functions. The ECRYPT_encrypt_bytes() function 
* (declared here) encrypts byte strings of arbitrary length, while 
* the ECRYPT_encrypt_blocks() function (defined later) only accepts 
* lengths which are multiples of ECRYPT_BLOCKLENGTH.
*
* The user is allowed to make multiple calls to 
* ECRYPT_encrypt_blocks() to incrementally encrypt a long message, 
* but he is NOT allowed to make additional encryption calls once he 
* has called ECRYPT_encrypt_bytes() (unless he starts a new message 
* of course). For example, this sequence of calls is acceptable:
*
* ECRYPT_keysetup();
*
* ECRYPT_ivsetup(); 
* ECRYPT_encrypt_blocks();
* ECRYPT_encrypt_blocks();
* ECRYPT_encrypt_bytes();
*
* ECRYPT_ivsetup();
* ECRYPT_encrypt_blocks();
* ECRYPT_encrypt_blocks();
*
* ECRYPT_ivsetup();
* ECRYPT_decrypt_bytes();
*
* The following sequence is not:
*
* ECRYPT_keysetup();
* ECRYPT_ivsetup();
* ECRYPT_encrypt_blocks();
* ECRYPT_encrypt_bytes();
* ECRYPT_encrypt_blocks();
*
* By default ECRYPT_encrypt_bytes() and ECRYPT_decrypt_bytes() are 
* defined as macros which redirect the call to a single function 
* ECRYPT_process_bytes(). If you want to provide separate encryption 
* and decryption functions, please undef 
* ECRYPT_HAS_SINGLE_BYTE_FUNCTION.
*/
#define ECRYPT_HAS_SINGLE_BYTE_FUNCTION       /* [edit] */
ifdef ECRYPT_HAS_SINGLE_BYTE_FUNCTION

#define ECRYPT_encrypt_bytes(ctx, plaintext, ciphertext, msglen)   
    ECRYPT_process_bytes(0, ctx, plaintext, ciphertext, msglen)
#define ECRYPT_decrypt_bytes(ctx, ciphertext, plaintext, msglen)   \
ECRYPT_process_bytes(1, ctx, ciphertext, plaintext, msglen)

void ECRYPT_process_bytes(
    int action,                 /* 0 = encrypt; 1 = decrypt; */
    ECRYPT_ctx* ctx,
    const u8* input,
    u8* output,
    u32 msglen);                /* Message length in bytes. */

#else

void ECRYPT_encrypt_bytes(
    ECRYPT_ctx* ctx,
    const u8* plaintext,
    u8* ciphertext,
    u32 msglen);                /* Message length in bytes. */

void ECRYPT_decrypt_bytes(
    ECRYPT_ctx* ctx,
    const u8* ciphertext,
    u8* plaintext,
    u32 msglen);                /* Message length in bytes. */

#endif

/* Optional features */

/* For testing purposes it can sometimes be useful to have a function
* which immediately generates keystream without having to provide it
* with a zero plaintext. If your cipher cannot provide this function
* (e.g., because it is not strictly a synchronous cipher), please
* reset the ECRYPT_GENERATES_KEYSTREAM flag.
*/

#define ECRYPT_GENERATES_KEYSTREAM
#ifdef ECRYPT_GENERATES_KEYSTREAM

void ECRYPT_keystream_bytes(
    ECRYPT_ctx* ctx,
    u8* keystream,
    u32 length);                /* Length of keystream in bytes. */

#endif

/* Optional optimizations */
/ * By default, the functions in this section are implemented using 
  * calls to functions declared above. However, you might want to 
  * implement them differently for performance reasons. 
  */ 
/* 
 * All-in-one encryption/decryption of (short) packets. 
 * 
 * The default definitions of these functions can be found in 
 * "ecrypt-sync.c". If you want to implement them differently, please 
 * undef the ECRYPT_USES_DEFAULT_ALL_IN_ONE flag. 
 */ 
#define ECRYPT_USES_DEFAULT_ALL_IN_ONE /* [edit] */ 
/* 
 * Undef ECRYPT_HAS_SINGLE_PACKET_FUNCTION if you want to provide 
 * separate packet encryption and decryption functions. 
 */ 
#define ECRYPT_HAS_SINGLE_PACKET_FUNCTION /* [edit] */
#define ECRYPT_HAS_SINGLE_PACKET_FUNCTION

#define ECRYPT_encrypt_packet(
  ctx, iv, plaintext, ciphertext, mglen)
  ECRYPT_process_packet(0, 
  ctx, iv, plaintext, ciphertext, mglen)

#define ECRYPT_decrypt_packet(
  ctx, iv, ciphertext, plaintext, mglen)
  ECRYPT_process_packet(1, 
  ctx, iv, ciphertext, plaintext, mglen)

void ECRYPT_process_packet(
  int action, /* 0 = encrypt; 1 = decrypt; */
  ECRYPT_ctx* ctx,
  const u8* iv,
  const u8* input,
  u8* output,
  u32 msglen);

#else

void ECRYPT_encrypt_packet(
  ECRYPT_ctx* ctx,
  const u8* iv,
  const u8* plaintext,
  u8* ciphertext,
  u32 msglen);

void ECRYPT_decrypt_packet(
  ECRYPT_ctx* ctx,
  const u8* iv,
const u8* ciphertext,
    u8* plaintext,
    u32 msglen);

#endif

/*
 * Encryption/decryption of blocks.
 *
 * By default, these functions are defined as macros. If you want to
 * provide a different implementation, please undef the
 * ECRYPT_USES_DEFAULT_BLOCK_MACROS flag and implement the functions
 * declared below.
 */

#define ECRYPT_BLOCKLENGTH 64                 /* [edit] */
#define ECRYPT_USES_DEFAULT_BLOCK_MACROS      /* [edit] */
#ifdef ECRYPT_USES_DEFAULT_BLOCK_MACROS

#define ECRYPT_encrypt_blocks(ctx, plaintext, ciphertext, blocks)  
    ECRYPT_encrypt_bytes(ctx, plaintext, ciphertext,                 
    (blocks) * ECRYPT_BLOCKLENGTH)

#define ECRYPT_decrypt_blocks(ctx, ciphertext, plaintext, blocks)  
    ECRYPT_decrypt_bytes(ctx, ciphertext, plaintext,                 
    (blocks) * ECRYPT_BLOCKLENGTH)

#ifdef ECRYPT_GENERATES_KEYSTREAM

#define ECRYPT_keystream_blocks(ctx, keystream, blocks)            
    ECRYPT_keystream_bytes(ctx, keystream,                           
    (blocks) * ECRYPT_BLOCKLENGTH)

#endif
#else

/* Undef ECRYPT_HAS_SINGLE_BLOCK_FUNCTION if you want to provide
 * separate block encryption and decryption functions.
 */
#define ECRYPT_HAS_SINGLE_BLOCK_FUNCTION      /* [edit] */
#ifdef ECRYPT_HAS_SINGLE_BLOCK_FUNCTION

#define ECRYPT_encrypt_blocks(ctx, plaintext, ciphertext, blocks)     
    ECRYPT_process_blocks(0, ctx, plaintext, ciphertext, blocks)

#define ECRYPT_decrypt_blocks(ctx, ciphertext, plaintext, blocks)     
    ECRYPT_process_blocks(1, ctx, ciphertext, plaintext, blocks)

void ECRYPT_process_blocks(
int action,                 /* 0 = encrypt; 1 = decrypt; */
ECRYPT_ctx* ctx,
const u8* input,
u8* output,
u32 blocks);       /* Message length in blocks. */

#else

void ECRYPT_encrypt_blocks(
    ECRYPT_ctx* ctx,
    const u8* plaintext,
    u8* ciphertext,
    u32 blocks);            /* Message length in blocks. */

void ECRYPT_decrypt_blocks(
    ECRYPT_ctx* ctx,
    const u8* ciphertext,
    u8* plaintext,
    u32 blocks);            /* Message length in blocks. */

#endif

#ifdef ECRYPT_GENERATES_KEYSTREAM

void ECRYPT_keystream_blocks(
    ECRYPT_ctx* ctx,
    u8* keystream,
    u32 blocks);            /* Keystream length in blocks. */

#endif

/*
* If your cipher can be implemented in different ways, you can use
* the ECRYPT_VARIANT parameter to allow the user to choose between
* them at compile time (e.g., gcc -DECRYPT_VARIANT=3 ...). Please
* only use this possibility if you really think it could make a
* significant difference and keep the number of variants
* (ECRYPT_MAXVARIANT) as small as possible (definitely not more than
* 10). Note also that all variants should have exactly the same
* external interface (i.e., the same ECRYPT_BLOCKLENGTH, etc.).
*/
#define ECRYPT_MAXVARIANT 1             /* [edit] */

#ifndef ECRYPT_VARIANT
#define ECRYPT_VARIANT 1
#endif

#if (ECRYPT_VARIANT > ECRYPT_MAXVARIANT)
#error this variant does not exist
#endif
#endif

#include <stdio.h>
#include <stdlib.h>
#include <openssl/evp.h>
#include "ecrypt-sync.h"

int aes_decrypt(EVP_CIPHER_CTX *e, unsigned char *ciphertext, unsigned char *plaintext, unsigned int len)
{
    int updatelen = len;
    int finallen = 0;

    EVP_DecryptInit_ex(e, NULL, NULL, NULL, NULL);
    EVP_DecryptUpdate(e, plaintext, &updatelen, ciphertext, len);
    EVP_DecryptFinal_ex(e, plaintext+updatelen, &finallen);

    return (updatelen + finallen);
}

void ECRYPT_init(void)
{
    /*nothing to do*/
}

void ECRYPT_keysetup(
    ECRYPT_ctx* ctx,
    const u8* key,
    u32 keysize, /* Key size in bits. */
    u32 ivsize) /* IV size in bits. */
{
    EVP_CIPHER_CTX_init(&ctx->aesctx);
    EVP_DecryptInit_ex(&ctx->aesctx, EVP_aes_128_ecb(), NULL, key, NULL);
    EVP_CIPHER_CTX_set_padding(&ctx->aesctx, 0);
}

void ECRYPT_ivsetup(
    ECRYPT_ctx* ctx,
    const u8* iv)
{
    __int128_t *ptr = (__int128_t*)iv;
    ctx->ctr = *ptr;
}

void ECRYPT_process_bytes(
    int action, /* 0 = encrypt; 1 = decrypt */
ECRYPT_ctx* ctx,
const u8* input,
u8* output,
u32 msglen) /* Message length in bytes. */
{
    unsigned int bytesLeft = msglen;
    unsigned char *ctr = (unsigned char*)&ctx->ctr;
    unsigned char pt[16];
    int msgIndex = 0;
    int i = 0;

    while( bytesLeft )
    {
        aes_decrypt(&ctx->aesctx, ctr, pt, 16);

        if( bytesLeft < 16 )
        {
            for(i = 0; i < bytesLeft; i++)
            {
                output[msgIndex] = pt[i] ^ input[msgIndex];
                msgIndex++;
            }
            bytesLeft = 0;
        }
        else
        {
            for(i = 0; i < 16; i++)
            {
                output[msgIndex] = pt[i] ^ input[msgIndex];
                msgIndex++;
            }
            bytesLeft -= 16;
        }
    }

    ctx->ctr++;
}
## APPENDIX C

### System Configuration

CPU:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>processor</td>
<td>0</td>
</tr>
<tr>
<td>vendor_id</td>
<td>GenuineIntel</td>
</tr>
<tr>
<td>cpu family</td>
<td>6</td>
</tr>
<tr>
<td>model</td>
<td>15</td>
</tr>
<tr>
<td>model name</td>
<td>Intel(R) Core(TM)2 Duo CPU T5550 @ 1.83GHz</td>
</tr>
<tr>
<td>stepping</td>
<td>13</td>
</tr>
<tr>
<td>cpu MHz</td>
<td>1833.000</td>
</tr>
<tr>
<td>cache size</td>
<td>2048 KB</td>
</tr>
<tr>
<td>physical id</td>
<td>0</td>
</tr>
<tr>
<td>siblings</td>
<td>2</td>
</tr>
<tr>
<td>core id</td>
<td>0</td>
</tr>
<tr>
<td>cpu cores</td>
<td>2</td>
</tr>
<tr>
<td>apicid</td>
<td>0</td>
</tr>
<tr>
<td>initial apicid</td>
<td>0</td>
</tr>
<tr>
<td>fpu</td>
<td>yes</td>
</tr>
<tr>
<td>fpu_exception</td>
<td>yes</td>
</tr>
<tr>
<td>cpuId level</td>
<td>10</td>
</tr>
<tr>
<td>wp</td>
<td>yes</td>
</tr>
<tr>
<td>flags</td>
<td>fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx lm constant_tsc arch_perfmon pebs bts rep_good aperfmperf pni dtes64 monitor ds_cpl est tm2 ssse3 cx16 xatmi pdcm lahf_lm dts</td>
</tr>
<tr>
<td>bogomips</td>
<td>3657.59</td>
</tr>
<tr>
<td>clflush size</td>
<td>64</td>
</tr>
<tr>
<td>cache_alignment</td>
<td>64</td>
</tr>
<tr>
<td>address sizes</td>
<td>36 bits physical, 48 bits virtual</td>
</tr>
</tbody>
</table>

Power management:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>processor</td>
<td>1</td>
</tr>
<tr>
<td>vendor_id</td>
<td>GenuineIntel</td>
</tr>
<tr>
<td>cpu family</td>
<td>6</td>
</tr>
<tr>
<td>model</td>
<td>15</td>
</tr>
<tr>
<td>model name</td>
<td>Intel(R) Core(TM)2 Duo CPU T5550 @ 1.83GHz</td>
</tr>
<tr>
<td>stepping</td>
<td>13</td>
</tr>
<tr>
<td>cpu MHz</td>
<td>1833.000</td>
</tr>
<tr>
<td>cache size</td>
<td>2048 KB</td>
</tr>
<tr>
<td>physical id</td>
<td>0</td>
</tr>
<tr>
<td>siblings</td>
<td>2</td>
</tr>
<tr>
<td>core id</td>
<td>1</td>
</tr>
<tr>
<td>cpu cores</td>
<td>2</td>
</tr>
<tr>
<td>apicid</td>
<td>1</td>
</tr>
<tr>
<td>initial apicid</td>
<td>1</td>
</tr>
<tr>
<td>fpu</td>
<td>yes</td>
</tr>
<tr>
<td>fpu_exception</td>
<td>yes</td>
</tr>
</tbody>
</table>
cpuid level : 10
wp : yes
flags : fpu vme de pse tsc mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx lm constant_tsc arch_perfmon pebs bts rep_good aperfmperf pni dtes64 monitor ds_cpl est tm2 ssse3 cx16 xtrm pdcmt lahf_lm dts
bogomips : 3657.52
clflush size : 64
cache_alignment : 64
address sizes : 36 bits physical, 48 bits virtual
power management:

COMPILER:

Using built-in specs.
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 4.4.4-14ubuntu5' --with-arch=amd64 --with-docdir=/usr/share/doc/gcc- 4.4/README.Bugs --enable-languages=c,c++,fortran,objc,obj-c++ -- prefix=/usr --program-suffix=--4.4 --enable-shared --enable-multilib -- enable-linker-build-id --with-system-zlib --libexecdir=/usr/lib -- without-include-gettext --enable-threads=posix --with-gxx--include- dir=/usr/include/c++/4.4 --libdir=/usr/lib --enable-nls --with- sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-objc- gc --disable-werror --with-arch-32=i686 --with-tune=generic --enable- checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu -- target=x86_64-linux-gnu
Thread model: posix
gcc version 4.4.5 (Ubuntu/Linaro 4.4.4-14ubuntu5)

COMPILATION:

make var=1 conf=gcc_default_default

gcc -g -Wall -pedantic -ansi -DECRYPT_API=encrypt-sync.h -
DECRYPT_VARIANT=1 -I../aes-ctr/..\include -I../aes-ctr -c -o encrypt-
test_gcc_default.o ..aes-ctr/..test/encrypt-test.c
gcc -g -Wall -pedantic -ansi -DECRYPT_API=encrypt-sync.h -
DECRYPT_VARIANT=1 -I../aes-ctr/..\include -I../aes-ctr -c -o encrypt-
sync_gcc_default.o ..aes-ctr/..api/encrypt-sync.c
gcc -g -Wall -pedantic -ansi -DECRYPT_API=encrypt-sync.h -
DECRYPT_VARIANT=1 -I../aes-ctr/..\include -I../aes-ctr -c -o aes-
ctr_gcc_default.o ..aes-ctr/aes-ctr.s
gcc -g -Wall -pedantic -ansi encrypt-test_gcc_default.o encrypt-
sync_gcc_default.o aes-ctr_gcc_default.o -o encrypt-test
REFERENCES


