Efficient Implementations of Cryptographic Algorithms

Shelley B. Kandola

July 24, 2012

Contents

1 Introduction 2
  1.1 The Programming Environment 2

2 Mathematical Background 2
  2.1 Addition 3
  2.2 Multiplication 3
    2.2.1 Multiplication by $x$ 4
    2.2.2 Multiplicative Inverses 4
    2.2.3 Logarithms and Exponentiation 4

3 AES Specifications 5
  3.1 Overview 5
  3.2 SubBytes 6
  3.3 ShiftRows 6
  3.4 MixColumns 7
  3.5 AddRoundKey 7
    3.5.1 Key Expansion 7

4 Programming Results 8
  4.1 Standard Implementation 8
  4.2 Four-Table Implementation 8
  4.3 One-Table Implementation 9
  4.4 Small-Table Implementation 11
  4.5 Conclusion 11

5 Countermeasure Results 12
  5.1 Introduction 12
  5.2 Masked SBox 13

6 Projected Research 14

References 14
1 Introduction

This Research Experience for Undergraduates was funded by the National Science Foundation and the Department of Defense from May 21, 2012 to July 27, 2012. The goals of the REU were to analyze efficient implementations of cryptographic algorithms and to research and implement countermeasures to side-channel attacks. The cryptographic algorithm on which this research was focused is the Advanced Encryption Standard, AES. Programming goals included implementing AES multiple ways, including as specified in the standard, with four lookup tables, and with one lookup table in both C and Assembly Language. These implementations were analyzed for efficiency. The latter half of the summer was spent strengthening AES against side-channel attacks using masking techniques.

The purpose of the paper is to explain the AES, to explore various implementations of AES, and to implement different countermeasures. This paper will then highlight the differences in efficiencies between each of the implementations.

1.1 The Programming Environment

The code programmed during the duration of this project was created in Code Composer Studio v3.3, by Texas Instruments. The environment was set up on a simulated little-endian Digital Signaling Processor: a C674x CPU Cycle Accurate Simulator with a TMS320C6400_0 CPU. Code Composer Studio offers a profiling feature equipped with a clock and other functions. This tool was used to measure the CPU cycles for various symbol names during the execution of a program.

2 Mathematical Background

All bytes used in the algorithm can be treated as elements of a finite field. Specifically, a Galois Field of order 256, denoted $GF(2^8)$. A Galois Field must have a prime-powered order and all of the properties of a field, that is; additive associativity, commutativity, identity element, and inverses; multiplicative associativity, commutativity, identity, and inverses; and the distributive property. Consequently, the elements of $GF(2^8)$ under both $+$ and $\cdot$ form Abelian groups. In $GF(2^8)$, bytes can be represented as polynomials. Each polynomial can have up to 8 terms and can have degree as high as seven, and every coefficient is in $\mathbb{Z}_2$, that is, either 1 or 0. For example, the byte 1100 1101 can be represented as $x^7 + x^6 + x^3 + x^2 + 1$. Below is a detailing of the properties of $GF(2^8)$, denoted $F$.

- Binary Operations
  - Addition, $+$
  - Multiplication, $\cdot$
- Additive Properties
  - Identity: $\exists e \in F \text{ s.t. } a + e = a \forall a \in F$
  - Inverses: $\forall a \in F, \exists -a \text{ s.t. } -a + a = e$
  - Closure: $a + b \in F \forall a, b \in F$
  - Associativity: $a + (b + c) = (a + b) + c \forall a, b, c \in F$
– Commutativity: \( a + b = b + a \forall a, b \in F \)

• Multiplicative Properties
  – Identity: \( \exists e \in F \text{ s.t. } a \cdot e = a \forall a \in F \)
  – Inverses: \( \forall a \in F, \exists a^{-1} \text{ s.t. } a \cdot a^{-1} = e \)
  – Closure: \( a \cdot b \in F \forall a, b \in F \)
  – Associativity: \( a \cdot (b \cdot c) = (a \cdot b) \cdot c \forall a, b, c \in F \)
  – Commutativity: \( a \cdot b = b \cdot a \forall a, b \in F \)
  – Distributivity: \( a \cdot (b + c) = (a \cdot b) + (a \cdot c) \forall a, b, c \in F \)

2.1 Addition

Because the only coefficients in \( GF(2^8) \) are 1 and 0, addition of polynomials is done modulo 2. For example, \( 1 + 1 = 0 \), \( 1 + 0 = 1 \), \( 0 + 0 = 0 \). This can be implemented using XOR. The addition of polynomials can be represented as \( \sum_{i=0}^{7} a_i x^i + \sum_{i=0}^{7} b_i x^i = \sum_{i=0}^{7} (a_i \oplus b_i) x^i \). The following equations are all equivalent:

\[
(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2
\]

\[
0101 0111 \oplus 1000 0011 = 11010100
\]

\[
0x57 \oplus 0x63 = 0xD4
\]

2.2 Multiplication

Multiplication is more complicated. Because polynomials in \( GF(2^8) \) cannot exceed eight terms, multiplication must be done modulo an irreducible polynomial of degree 8. An irreducible polynomial is a polynomial that cannot be factored. For AES, this irreducible polynomial is \( m(x) = x^8 + x^4 + x^3 + x + 1 \). In implementations, this is represented by the hexadecimal number \( 0x11B \). A polynomial \( p(x) \text{ mod } m(x) \) can be computed by calculating \( p(x) \oplus m(x) \). Consider the following example:

\[
0x57 \cdot 0x83 = 0x15BD
\]

\[
0x15BD \text{ mod } 0x11B = 0x15BD \text{ XOR } 11B = 0xC1
\]

When converted to polynomials in \( GF(2^8) \), this becomes:

\[
(x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1) = x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1
\]

\[
x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \mod x^8 + x^4 + x^3 + x + 1 = x^7 + x^6 + 1
\]

The following is a method of performing \( GF(2^8) \) multiplication in C.

```c
#define GFm(x,y)
    (((y & 1) * x) ^
     (((y>>1) & 1) * xtime(x)) ^
     (((y>>2) & 1) * xtime(xtime(x))) ^
     (((y>>3) & 1) * xtime(xtime(xtime(x)))) ^
     (((y>>4) & 1) * xtime(xtime(xtime(xtime(x))))) & 0xff)
```

3
Because multiplication in $GF(2^8)$ is different than multiplication in $\mathbb{Z}$, it does not suffice to implement multiplication using just $\cdot$. The method above involves splitting the multiplicand into individual bits, multiplying the bit by the multiplier shifted left by the appropriate amount, then XORing all of the results together, followed by an AND with 0xFF. This is done through nested applications of xtime(), which is detailed below.

### 2.2.1 Multiplication by $x$

Many implementations of AES include a more efficient method of multiplying polynomials by $x$. In hexadecimal, this becomes 0x02 and in binary, 0000 0010. If the polynomial $p(x)$ is of degree 6, i.e., if the coefficient $b_7$ is 0, then multiplication by $x$ is equivalent to shifting left by one bit. If $p(x)$ is of degree 7 and coefficient $b_7$ is 1, then multiplication by 2 can be computed by performing a left shift by one bit then an XOR with 0x1B. In implementations this method is often known as xtime(). Multiplication by constants greater than $x$ can be performed using repeated applications of xtime(). For example, 0x57 · 0x13 can be computed by calculating 0x57 · (0x01 ⊕ 0x02 ⊕ 0x10).

This method can be implemented in C using the following macro, where xtime() is detailed in the following section.

```c
#define xtime(x) ((x<<1) ^ (((x>>7) & 1) * 0x1B))
```

### 2.2.2 Multiplicative Inverses

Because $GF(2^8)$ is a field, all elements except for the additive identity 0 have multiplicative inverses. The multiplicative inverse of a polynomial $p(x)$ is denoted $p^{-1}(x)$ such that $p(x)p^{-1}(x) = 1$. The inverse polynomial can be found by extending the Euclidean algorithm. For each $p(x)$, there exists $q(x), c(x)$ such that $p(x)q(x) + m(x)c(x) = 1$. Hence $p^{-1}(x) = q(x) \mod m(x)$ since $p(x)q(x) \mod m(x) = 1$.

### 2.2.3 Logarithms and Exponentiation

Generators are elements of a field whose exponentiation can produce all of the elements in a given field. For $GF(2^8)$, 0x03 is one of those generators. Using this fact, tables can be generated that represent the logarithm and exponentiation tables of $GF(2^8)$. The rules are simple: $\exp(0) = 1$, $\exp(i) = \exp(i - 1) \cdot g$ and $\log(\exp(i)) = i$, where $g$ is the generator 0x03. These tables can be used to easily calculate inverse polynomials in $GF(2^8)$. Using logarithm/exponentiation notation division can be rewritten as such:

$$
\frac{p(x)}{q(x)} = \exp(\log(p(x)) - \log(q(x)))
$$

Hence finding the inverse of a polynomial $p(x)$ can be as simple as calculating $\exp(\log(p(x)) - \log(1))$ using the logarithm/exponentiation lookup tables. To use the tables, refer to the row number corresponding to byte 0, and the column referring to byte 1. According to this, $\log(0x01)$ is 0xff.
3 AES Specifications

3.1 Overview

The Advanced Encryption Standard encompasses the Rijndael algorithm. It is a symmetric block cipher that encrypts and decrypts 128-bit data blocks with a cipher key of either bit-length 128,
This paper will refer only to the version that uses a 128-bit cipher key, AES-128. To strengthen the cipher-key, it is expanded to be used across each round of the algorithm. The AES-128 comprises ten rounds. Each round comprises four functions: \textit{SubBytes()}, \textit{ShiftRows()}, \textit{MixColumns()}, and \textit{AddRoundKey()}; the 10th round is an exception and does not include \textit{MixColumns()}.

### 3.2 SubBytes

\textit{SubBytes()} is a simple method that replaces a byte \( j \) with the \( j \)th index of a substitution box. The substitution box, or SBox, contains 256 entries and uses .5KB of memory. Generating the SBox consists of two steps.

1. Take the multiplicative inverse of the element in \( GF(2^8) \)
2. Apply an affine transformation

Note that the multiplicative inverse of \( 0x00 \) is fixed to \( 0x00 \), which would otherwise not exist in the finite field \( GF(2^8) \). The affine transformation is a bit more complicated. To transform one byte, each bit must be changed such that for \( 0 \leq i < 8 \):

\[
b'_i = b_i \oplus b_{(i+4)} \mod 8 \oplus b_{(i+5)} \mod 8 \oplus b_{(i+6)} \mod 8 \oplus b_{(i+7)} \mod 8 \oplus c_i
\]

In AES, the byte \( c \) is the constant \( 0x63 \), or 0110 0011. To compute the value at the first index (index 0), apply its multiplicative inverse and XOR each bit with the \( i \)th bit of \( 0x63 \). Since we set \( 0x00 \) to be its own multiplicative inverse, the result is \( 0x63 \). The affine transformation can also be represented as matrix multiplication.

\[
\begin{bmatrix}
  b'_0 \\
  b'_1 \\
  b'_2 \\
  b'_3 \\
  b'_4 \\
  b'_5 \\
  b'_6 \\
  b'_7 \\
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
  1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
  1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
  1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
  0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
  0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
  0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
\end{bmatrix} \cdot \begin{bmatrix}
  b_0 \\
  b_1 \\
  b_2 \\
  b_3 \\
  b_4 \\
  b_5 \\
  b_6 \\
  b_7 \\
\end{bmatrix} + \begin{bmatrix}
  1 \\
  1 \\
  0 \\
  0 \\
  1 \\
  1 \\
  1 \\
  0 \\
\end{bmatrix}
\]

\textit{InvSubBytes()} works similarly. The inverse SBox is generated by first applying the affine transformation to the byte to be substituted, and then taking its multiplicative inverse in \( GF(2^8) \). As expected, \( \text{InvSBox}[63] = 0x00 \).

### 3.3 ShiftRows

The \textit{ShiftRows()} transformation cyclically shifts each row \( i \) of the state \( i \) bytes to the left. Consequently, the first row is not shifted. This can be generalized by the following:

\[
s'_{r,c} = s_{r,(c+r) \mod 4}
\]
The effect achieved is swapping bytes that appear higher in a row with bytes that appear lower in a row.

\[
\begin{pmatrix}
S_0,0 & S_0,1 & S_0,2 & S_0,3 \\
S_1,0 & S_1,1 & S_1,2 & S_1,3 \\
S_2,0 & S_2,1 & S_2,2 & S_2,3 \\
S_3,0 & S_3,1 & S_3,2 & S_3,3 \\
\end{pmatrix}
\Rightarrow
\begin{pmatrix}
S_0,0 & S_0,1 & S_0,2 & S_0,3 \\
S_1,1 & S_1,2 & S_1,3 & S_1,0 \\
S_2,2 & S_2,0 & S_2,1 & S_2,3 \\
S_3,3 & S_3,0 & S_3,1 & S_3,2 \\
\end{pmatrix}
\]

The InvShiftRows() function can be represented by a similar equation, \( s'_{r,c} = s_{r,(c-r) \mod 4} \).

### 3.4 MixColumns

The standard specifies that MixColumns() is a "transformation in the Cipher that takes all of the columns of the State and mixes their data (independently of one another) to produce new columns [1, p.11]." MixColumns() treats each column as a 4-term polynomial and multiplies it by \( a(x) = 0x03x^3 + 0x01x^2 + 0x01x + 0x02 \) modulo irreducible polynomial \( x^4 + 1 \). When expanded, this is equivalent to the following matrix multiplication.

\[
\begin{pmatrix}
S'_{0,c} \\
S'_{1,c} \\
S'_{2,c} \\
S'_{3,c} \\
\end{pmatrix}
= \begin{pmatrix}
2 & 3 & 1 & 1 \\
1 & 2 & 3 & 1 \\
1 & 1 & 2 & 3 \\
3 & 1 & 1 & 2 \\
\end{pmatrix}
\cdot
\begin{pmatrix}
S_{0,c} \\
S_{1,c} \\
S_{2,c} \\
S_{3,c} \\
\end{pmatrix}
= \begin{pmatrix}
S_{0,c} \cdot 2 \oplus S_{1,c} \cdot 3 \oplus S_{2,c} \oplus S_{3,c} \\
S_{0,c} \oplus S_{1,c} \cdot 2 \oplus S_{2,c} \cdot 3 \oplus S_{3,c} \\
S_{0,c} \oplus S_{1,c} \oplus S_{2,c} \cdot 2 \oplus S_{3,c} \cdot 3 \\
S_{0,c} \cdot 3 \oplus S_{1,c} \oplus S_{2,c} \oplus S_{3,c} \cdot 2 \\
\end{pmatrix}
\]

The decryption method InvMixColumns() functions similarly, although the fixed polynomial \( a(x) \) is replaced with \( a^{-1}(x) = 0x0Bx^3 + 0x0Dx^2 + 0x09x + 0x0E \). The polynomial \( a^{-1}(x) \) is the inverse of \( a(x) \) in \( GF(2^8) \) such that \( a(x)a^{-1}(x) = 1 \), where 1 is the multiplicative identity of the field. Note that the multiplication is the multiplication of hexadecimal numbers.

\[
\begin{pmatrix}
S'_{0,c} \\
S'_{1,c} \\
S'_{2,c} \\
S'_{3,c} \\
\end{pmatrix}
= \begin{pmatrix}
e & b & d & 9 \\
9 & e & b & d \\
d & 9 & e & b \\
b & d & 9 & e \\
\end{pmatrix}
\cdot
\begin{pmatrix}
S_{0,c} \\
S_{1,c} \\
S_{2,c} \\
S_{3,c} \\
\end{pmatrix}
= \begin{pmatrix}
S_{0,c} \cdot e \oplus S_{1,c} \cdot b \oplus S_{2,c} \cdot d \oplus S_{3,c} \cdot 9 \\
S_{0,c} \cdot 9 \oplus S_{1,c} \cdot e \oplus S_{2,c} \cdot b \oplus S_{3,c} \cdot d \\
S_{0,c} \cdot d \oplus S_{1,c} \cdot 9 \oplus S_{2,c} \cdot e \oplus S_{3,c} \cdot b \\
S_{0,c} \cdot b \oplus S_{1,c} \cdot d \oplus S_{2,c} \cdot 9 \oplus S_{3,c} \cdot e \\
\end{pmatrix}
\]

### 3.5 AddRoundKey

The AddRoundKey() transformation modifies the state by using the XOR operation to add a Round Key. The Round Keys are generated before the execution of the algorithm by the Key Expansion method detailed below. Before the ten rounds begin, the 128-bit key is split into four 32-bit words that are applied to the state using XOR.

#### 3.5.1 Key Expansion

One noted security feature of AES is the 128-bit key’s expansion to ten keys. Keys are applied column-wise using XOR to the state, so a total of 44 words need to be generated for the algorithm. The following is pseudo code modified from the standard to work for AES-128. It generates forty-four 32-bit words that are applied to each column of the state.

KeyExpansion()
begin
word temp
i=0
while (i<4)
    w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
    i=i+1
end while
i = 4
while (i<44)
    temp = w[i-1]
    if (i mod 4 = 0)
        temp = SubWord(RotWord(temp)) XOR Rcon[i/4]
    end if
    w[i] = w[i-4] XOR temp
    i=i+1
end while

The method SubWord() is a function that takes a 32-bit word, divides it into four bytes, and replaces each byte according to the SBox. It creates a word out of the four bytes and returns it. The method RotWord() rotates a 32-bit word left by one byte. For example, $[a_0, a_1, a_2, a_3]$ becomes $[a_1, a_2, a_3, a_0]$. Rcon[] refers to an array of constants such that the $i^{th}$ index of the array is the word created by $[x^{i-1}, 0x00, 0x00, 0x00]$, where $i > 0$ and $x$ is a polynomial represented by $0x02$ in hexadecimal.

4 Programming Results

4.1 Standard Implementation

The first two weeks of the summer were spent learning and understanding AES and implementing it according to the standard specified in FIPS-197. This implementation is very straightforward, as the pseudo code is specified directly in the standard. Consequently, the implementation is also inefficient. During the implementation, one change was made that differed it from the original specification. Instead of representing the state as a 4x4 2-dimensional array, the state was represented as a 4-item array of 32-bit columns comprising four 8-bit integers.

While it might seem less efficient to represent columns of the state this way because isolating bytes is more complicated than simply calling state[i][j], it is more efficient in assembly code. For example, a round key could be applied with 16 XORs in a straightforward C implementation, or it could be applied with 4 XORs when each column is its own integer.

By enabling the clock and profiling in Code Composer Studio, it is possible to determine the number of CPU cycles spent on a given symbol name in the code.

4.2 Four-Table Implementation

By using an additional 4KB of memory, AES can be made more efficient. The second implementation studied during this research project was a 4-table implementation. This implementation is more efficient because it replaces SubBytes(), ShiftRows(), and MixColumns() with a single table lookup. Consider the matrix used during the MixColumns() step.
Table 3: C674x Cycles for Standard C Implementation

<table>
<thead>
<tr>
<th>Implementation</th>
<th># of Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption</td>
<td>1702</td>
</tr>
<tr>
<td>Decryption</td>
<td>1647</td>
</tr>
</tbody>
</table>

Note that all multiplication is done in $GF(2^8)$ and $\oplus$ represents XOR. Let us consider the case for a column of the state whose first word is 0x00. When this is applied to the SBox, we get 0x63. This can be used to construct four tables that replace SubBytes(), ShiftRows(), and MixColumns(). Each table is made of entries that are the columns of the MixColumns() matrix multiplied by the table’s index’s SBox substitution. The first entry of the first table is $0xC66363A5$. Note that $3 \cdot 0x63$ is $0xA5$ because of the following:

$$3x = x(2 + 1)$$

$$3 \cdot 0x63 = 0x63(2 \oplus 1)$$

$$3 \cdot 0x63 = 0xC6 \oplus 0x63$$

$$3 \cdot 0x63 = 0xA5$$

This method is more efficient because it utilizes $\oplus$ instead of performing multiplication in $GF(2^8)$, which is a tedious process and computationally expensive. Because the coefficients of the matrix multiplication in InvMixColumns() are not 1, 2 and 3, however, $GF(2^8)$ multiplication must be implemented.

To generate the remaining three tables of the four-table implementation, one need only rotate each value of the previous table right one byte. The first entry of the second table is $0xA5C66363$. The remaining two tables are generated similarly. One column of a state during a round can be generated by taking the XOR of the the appropriately shifted bytes' indeces in corresponding tables. Consider the following C macro where a, b, c, d represent the already-shifted bytes.

```c
#define MIXCOL(a, b, c, d) \
(Te0[(a)] ^ Te1[(b)] ^ Te2[(c)] ^ Te3[(d)])
```

Decryption follows similarly, where each table is generated by the Inverse SBox.

Compared to the previous standard implementation, the 4-table implementation is much more efficient. Note, however, that 4KB of memory are being used to store the lookup tables.

### 4.3 One-Table Implementation

The one-table implementation of AES obviates tables 1-3, saving 3KB of memory. The remaining 1KB table serves the purpose the other three tables with simple rotations. Recall that $Te0[0x00] =$
Table 4: C674x Cycles for 4-Table Implementation

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption (asm)</td>
<td>138</td>
</tr>
<tr>
<td>Decryption (asm)</td>
<td>138</td>
</tr>
<tr>
<td>Decryption (C)</td>
<td>224</td>
</tr>
<tr>
<td>Decryption (C)</td>
<td>244</td>
</tr>
</tbody>
</table>

0xC66363A5, Te1[0x00] = 0xA5C66363, Te2[0x00] = 0x63A5C663, and Te3[0x00] = 0x6363A5C6.
Note how each entry is an 8-bit rotation of the corresponding entry in the previous table. Using this fact, values from the last three tables can be generated using only the first table and a rotation.
Below is a C implementation of the critical function.

```c
#define MIXCOL(a, b, c, d) \
(Te0[(a)] ^ ROL(Te0[(b)],24) ^ ROL(Te0[(c)],16) ^ ROL(Te0[(d)],8))
```

Variables a, b, c, d represent the already shifted bytes, and ROL() is a function that takes a 32-bit word and an amount of bits by which to rotate the word left. While this saves 3KB of memory, it is slightly less computationally efficient, however, easier to implement on devices where space is limited. To implement the one-table version is assembly, an Assembly Schedule was laid out that organized all the function calls within a round.

Table 5: Assembly Schedule for 1-Table AES Round

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XOR 10,13</td>
<td>EXT s2[2]</td>
<td>LDW 00</td>
<td>XOR 30,33</td>
<td>EXT s0[2]</td>
<td>LDW 20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>EXT s1[1]</td>
<td>LDW 02</td>
<td>EXT s3[1]</td>
<td>LDW 22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SHR s1[0]0</td>
<td>LDW 03</td>
<td>SHR s3[0]</td>
<td>LDW 23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>EXT s2[1]1</td>
<td>LDW 10</td>
<td>EXT s0[1]</td>
<td>LDW 30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>EXT s0[2]</td>
<td>LDW 11</td>
<td>EXT s1[2]</td>
<td>LDW 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>AND s0[3]</td>
<td>ROT 02,16</td>
<td>LDW 12</td>
<td>AND s2[3]</td>
<td>ROT 22,16</td>
<td>LDW 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>XOR k0,00</td>
<td>ROT 01,24</td>
<td>LDW 13</td>
<td>XOR k2,20</td>
<td>ROT 21,24</td>
<td>LDW 33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>XOR 00,02</td>
<td>BDEC</td>
<td>ROT 03,8</td>
<td>LDW k0</td>
<td>XOR 20,22</td>
<td>ROT 23,8</td>
<td>LDW k2</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>XOR 00,01</td>
<td>LDW k1</td>
<td>XOR 20,21</td>
<td>LDW k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>XOR 00,03</td>
<td>ROT 11,24</td>
<td>XOR 20,23</td>
<td>ROT 31,24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>XOR k1,10</td>
<td>ROT 12,16</td>
<td>XOR k3,30</td>
<td>ROT 32,16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>XOR 10,11</td>
<td>SHR s0[0]</td>
<td>ROT 13,8</td>
<td>XOR 30,31</td>
<td>SHR s2[0]</td>
<td>ROT 33,8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>XOR 10,12</td>
<td></td>
<td></td>
<td></td>
<td>XOR 30,32</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Because of the delay slots required for some assembly instructions, subsequent instructions have to be spaced out until the data they use is made available. The column headers refer to the units of the C674x processor. Different assembly instructions can be performed in different units based on their complexity.
Table 6: C674x Cycles for 1-Table Implementation

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption (asm)</td>
<td>156</td>
</tr>
<tr>
<td>Decryption (asm)</td>
<td>156</td>
</tr>
<tr>
<td>Encryption (C)</td>
<td>252</td>
</tr>
<tr>
<td>Decryption (C)</td>
<td>261</td>
</tr>
</tbody>
</table>

4.4 Small-Table Implementation

The small-table implementation eliminates the use of precomputed tables altogether, using 4KB less space than the 4-table implementation. Instead, these tables are effectively generated on-the-go using the SBox. The process for generating a temporary table value involves getting the SBox substitution for the byte, repeating the byte until it’s 32 bites long, performing byte-wise multiplication in $GF(2^8)$ with the column coefficients of $\text{MixColumns}()$, then $\text{XOR}$ing the result into the new column.

In C, the simplest way to do this is to multiply each byte by its corresponding $\text{MixColumns}()$ coefficient, then shift and $\text{XOR}$ the results together. In assembly, extending one byte to 32 bits becomes more efficient through the use of $\text{PACK2}$ and $\text{PACKL4}$. The instruction $\text{PACK2}$ takes the last 2 bytes of the two sources and stores them in a register. For example, the instruction $\text{PACK2 .S1 A.00, A.00, A.00}$ takes the bytes 2 and 3 from $A.00$ and copies them over to bytes 0 and 1. This followed by $\text{PACKL4 .L1 A.00, A.00, A.00}$ fills all four bytes of register $A.00$ with what was originally in byte 3. This takes two instructions, whereas repeated shifting and $\text{XOR}$ing would take six.

To complete generating the veritable table entry, the register needs to be multiplied by the appropriate coefficients. If encrypting a byte from the first row, those coefficients would be 0x02010103. The $\text{GMPY4}$ instruction performs byte-wise multiplication between two registers in the Galois Field specified by the control register $\text{GFPGFR}$, which stands for Galois Field Polynomial Generator Function Register. To simulate $GF(2^8)$, the value 0x0700001B needs to be moved into it. The field size is represented by 0x7 and the irreducible polynomial is represented by 0x1B. The remaining bits are ignored and reserved for other uses.

Note how while the assembly implementations run in similar time, the C decryption implementation runs much slower than the encryption implementation. Encryption in C utilizes the $\text{xttime()}$ method, whereas the previously mentioned $\text{GFm(x,y)}$ was implemented for decryption.

4.5 Conclusion

In conclusion, all implementations except for the standard have their own benefits in certain situations. Recall that there was no standard implementation done in assembly because it would have been very inefficient. The standard implementation in C was solely for the person of learning and understanding the algorithm as-is.
Table 7: Assembly Schedule for Small-Table AES Round

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XOR 10,13</td>
<td>EXT s2[2]</td>
<td>LDB 00</td>
<td>XOR 30,33</td>
<td>EXT s0[2]</td>
<td>LDB 20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>AND s[3]</td>
<td>LDB 03</td>
<td>SHR s[0]</td>
<td>LDB 10</td>
<td>EXT s[1]</td>
<td>LDB 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>PK2 00</td>
<td>LDB 10</td>
<td>PK2 20</td>
<td>LDB 32</td>
<td>PK2 20</td>
<td>LDB 33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>PK2 02</td>
<td>LDB 13</td>
<td>PK2 22</td>
<td>GMP 20</td>
<td>PK2 21</td>
<td>GMP 22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>PK2 03</td>
<td>GMP 00</td>
<td>PK2 23</td>
<td>GMP 23</td>
<td>PK2 23</td>
<td>GMP 21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>PK2 10</td>
<td>GMP 01</td>
<td>PK2 24</td>
<td>PK2 30</td>
<td>PK2 30</td>
<td>GMP 21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>PK2 11</td>
<td>GMP 03</td>
<td>LDW k0</td>
<td>PK2 31</td>
<td>PK2 31</td>
<td>GMP 23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>PK2 12</td>
<td>GMP 10</td>
<td>XOR k0</td>
<td>PK2 32</td>
<td>PK2 32</td>
<td>XOR k2,20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>PK2 13</td>
<td>GMP 11</td>
<td>XOR 00</td>
<td>PK2 33</td>
<td>PK2 33</td>
<td>XOR 20,22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>PK2 14</td>
<td>BDEC</td>
<td>GMP 12</td>
<td>XOR 00</td>
<td>PK4 13</td>
<td>BDEC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>PK2 15</td>
<td>GMP 13</td>
<td>XOR 00</td>
<td>GMP 33</td>
<td>XOR 20,23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>PK2 16</td>
<td>LDW k1</td>
<td>PK2 34</td>
<td>GMP 33</td>
<td>XOR 20,23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>PK2 17</td>
<td>XOR k1</td>
<td>LDW k3</td>
<td>XOR k3</td>
<td>XOR k3,30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>PK2 18</td>
<td>XOR 11</td>
<td>SHR s[0]</td>
<td>XOR 30</td>
<td>XOR 30,31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>PK2 19</td>
<td>XOR 12</td>
<td>SHR s[2]</td>
<td>XOR 30</td>
<td>XOR 30,32</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8: C674x Cycles for Small-Table Implementation

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption (asm)</td>
<td>212</td>
</tr>
<tr>
<td>Decryption (asm)</td>
<td>211</td>
</tr>
<tr>
<td>Encryption (C)</td>
<td>457</td>
</tr>
<tr>
<td>Decryption (C)</td>
<td>2412</td>
</tr>
</tbody>
</table>

Table 9: C674x Cycles for All Implementations

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Standard</th>
<th>4-Table</th>
<th>1-Table</th>
<th>5-Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>C</td>
<td>ASM</td>
<td>C</td>
<td>ASM</td>
</tr>
<tr>
<td>Encryption</td>
<td>1702</td>
<td>N/A</td>
<td>224</td>
<td>138</td>
</tr>
<tr>
<td>Decryption</td>
<td>1647</td>
<td>N/A</td>
<td>244</td>
<td>138</td>
</tr>
</tbody>
</table>

5 Countermeasure Results

5.1 Introduction

While mathematically secure, implementations of AES are vulnerable to side-channel attacks. One method used as a countermeasure for the Differential Power Analysis (DPA) attack is masking. The
goal of masking is to ensure that the data is never isolated. Typically this is done by XORing a mask, performing a transformation, then XORing the mask off again. This is especially effective when the transformation is linear, such as \texttt{ShiftRows()}, \texttt{MixColumns()}, or \texttt{AddRoundKey()}. \texttt{SubBytes()}, however, is non-linear and therefore difficult to mask. Masking efforts focus on the inversion step of \texttt{SubBytes()}, where $SBox(x) = x^{-1} + Af$, where $Af$ is the affine transformation.

By the analog of Fermat’s little theorem, we have a simpler way of computing the multiplicative inverse of an element than using log/exp lookup tables. We have that

$$a^{p-1} \equiv 1 \mod p \Rightarrow a^{-1} = a^{p-2}$$

where $p$ is the order of the field. Following Fermat’s little theorem, we know that a field $\mathbb{F}$ of order $p$ forms a group under multiplication of order $p-1$ (the nonzero elements). For $x \in \mathbb{F}(2^8)$, this means $x^{-1}$ can be calculated by $x^{254}$.

\textbf{Remark 1} $\mathbb{F}(2^8)$ is a field of order 256 under addition and multiplication. As a group under multiplication, it has order 255 as the zero element is removed. Hence $x^{-1} = x^{(2^8-1)-1} = x^{254}$.

### 5.2 Masked SBox

Because \texttt{SubBytes()} is weakest to DPA attacks, a version was implemented in which 32 different SBoxes were used to encrypt the plain text. For the first nine rounds and last round, each byte of the state gets its own SBox based on two randomly generated variables $m, n$. The SBoxes are generated such that for each $m, n$ pair, $SBox(i \oplus m) = SBox(i) \oplus n$. The pseudocode is as follows.

```c
int m[16]
int n[16]
int maskSBox[16][256]

for i from 0 to 16
    m[i] = random()
    n[i] = random()
    for j from 0 to 256
        maskSBox[i][j xor m[i]] = SBox[i] xor n[i]
    end for
end for
```

While generating the specific SBoxes is a rather simple task, removing the masks is not. Below is a representation of the state after the first three steps of the round have been applied: \texttt{SubBytes()}, \texttt{ShiftRows()}, and \texttt{MixColumns()}. Note that this is without masking. To unmask the state, a 16-byte array is needed that stores the unmask variable. Each byte can be computed as such:

$$B = C_0(S_0 \oplus n_0) \oplus C_1(S_1 \oplus n_1) \oplus C_2(S_2 \oplus n_2) \oplus C_3(S_3 \oplus n_3)$$

where 0,1,2,3 refer to the bytes in the column of the byte for which the unmasking variable is being generated.
Table 10: AES State after SubBytes(), ShiftRows(), MixColumns()

| $S_0 \cdot 2 \oplus S_5 \cdot 3 \oplus S_{10} \oplus S_{15}$ | $S_4 \cdot 2 \oplus S_9 \cdot 3 \oplus S_{14} \oplus S_3$ | $S_8 \cdot 2 \oplus S_{13} \cdot 3 \oplus S_2 \oplus S_7$ | $S_{12} \cdot 2 \oplus S_1 \cdot 3 \oplus S_6 \oplus S_{11}$ |
| $S_0 \oplus S_5 \cdot 2 \oplus S_{10} \cdot 3 \oplus S_{15}$ | $S_4 \oplus S_9 \cdot 2 \oplus S_{14} \cdot 3 \oplus S_3$ | $S_8 \oplus S_{13} \cdot 2 \oplus S_2 \cdot 3 \oplus S_7$ | $S_{12} \oplus S_1 \cdot 2 \oplus S_6 \cdot 3 \oplus S_{11}$ |
| $S_0 \oplus S_5 \oplus S_{10} \cdot 2 \oplus S_{15} \cdot 3$ | $S_4 \oplus S_9 \oplus S_{14} \cdot 2 \oplus S_3 \cdot 3$ | $S_8 \oplus S_{13} \oplus S_2 \cdot 2 \oplus S_7 \cdot 3$ | $S_{12} \oplus S_1 \oplus S_6 \cdot 2 \oplus S_{11} \cdot 3$ |
| $S_0 \cdot 3 \oplus S_5 \oplus S_{10} \oplus S_{15} \cdot 2$ | $S_4 \cdot 3 \oplus S_9 \oplus S_{14} \oplus S_3 \cdot 2$ | $S_8 \cdot 3 \oplus S_{13} \oplus S_2 \oplus S_7 \cdot 2$ | $S_{12} \cdot 3 \oplus S_1 \oplus S_6 \oplus S_{11} \cdot 2$ |

6 Projected Research

References

