Rijndael AES Implementation

Bibek Ghosh

Indian Statistical Institute, Kolkata

1 Introduction

A block is a sequence of bits of a given fixed length. A block cipher is a permutation that maps n-bits plaintext blocks to n-bit ciphertext blocks which is parameterized by a sequence of bits called the key. Formally, it is a map $E : K \times \{0,1\}^n \to \{0,1\}^n$, where for each $K \in \mathcal{K}$, the function $E_K(\cdot) \triangleq E(K, \cdot)$ is a bijection from $\{0,1\}^n$ to itself. A block cipher is a fundamental primitive in cryptography and is a major building block of several important cryptographic functionalities.
In 1997, NIST initiated the Advanced Encryption Standard (AES) development effort and called for the public to submit candidate algorithms for block ciphers. Only 15 AES candidates were accepted for the first evaluation round. After two open conferences, in 1999 NIST narrowed down the list to five candidates. In 2000, NIST announced the selection of Rijndael[Joa01] for the AES. AES became an official standard in November of 2001 as FIPS 197[NIS01b]. Block ciphers are the foundation for many cryptographic services, especially those that provide assurance of the confdentiality of data.

2 Preliminaries

2.1 Multiplication in $GF(2^8)$

In the algorithm of Rijndael there are no multiplications of two variables in GF $\left(2^8\right)$, but only the multiplication of a variable with a constant. The latter is easier to implement than the former. We describe here how multiplication by the value 02 can be implemented. The polynomial associated with 02 is x. Therefore, if we multiply an element b with 02 , we get:

$$\begin{aligned} b \cdot x =& b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x \pmod{m(x)} \\ =& b_6x^7 + b_5x^6 + b_4x^5 + (b_3 \oplus b_7)\,x^4 + (b_2 \oplus b_7)\,x^3 + b_1x^2 + (b_0 \oplus b_7)\,x + b_7 \end{aligned}$$

The multiplication by 02 is denoted `xtime`(x). `xtime`(x) can be implemented with a shift operation and a conditional XOR operation. To prevent timing attacks, attention must be paid so that `xtime`(x) is implemented in such a way that it takes a fixed number of cycles, independently of the value of its argument. This can be achieved by inserting dummy instructions at the right places. However, this approach is likely to introduce weaknesses against power analysis attacks.
We can (and should) do the reduction in constant time using masking. That is, instead of using the following code to do the reduction:

```
return (b << 1)^((b & 0x80) ? 0x1B : 0x00);
```

we can simply do:

```
return (b << 1)^((b >> 7) * 0x1B);
```

This can then be trivially bitsliced, e.g. like this:

```
//bit0..bit7 store the 8 bits of the result(s), bit8 is the overflow bit
bit0 ^= bit8;
bit1 ^= bit8;
bit3 ^= bit8;
bit4 ^= bit8;
```

A better approach seems to define a table M, where $M[a] = 02 \cdot a$. The routine `xtime`(x) is then implemented as a table look-up into M. Since all elements of GF $\left(2^8\right)$ can be written as a sum of powers of 02 , multiplication by any constant value can be implemented by a repeated use of `xtime`(x).
Example: The multiplication of an input b by the constant value 15 can be implemented as follows:

$$\begin{aligned} b \cdot 15 &= b \cdot (01 \oplus 04 \oplus 10) \\ &= b \cdot \left(01 \oplus 02^2 \oplus 02^4\right) \\ &= b \oplus \mathrm{xtime}(\mathrm{xtime}(b)) \oplus \mathrm{xtime}(\mathrm{xtime}(\mathrm{xtime}(\mathrm{xtime}(b)))) \\ &= b \oplus \mathrm{xtime}(\mathrm{xtime}(b \oplus \mathrm{xtime}(xtime(b)))) \end{aligned}$$

2.2 Decomposition of $\mathrm{S_{RD}}$

The Rijndael S-box $\mathrm{S_{RD}}$ is constructed from two transformations:

$$\mathrm{S_{RD}}[a] = f(g(a))$$

where $g(a)$ is the transformation

$$a \rightarrow a^{-1} \text{ in GF } \left(2^8\right)$$

and $f(a)$ is an affine transformation defined by:

$$b = f(a) \Longleftrightarrow \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

The transformation $g(a)$ is a self-inverse and hence

$$\mathrm{S_{RD}}^{-1}[a] = g^{-1}\left(f^{-1}(a)\right) = g\left(f^{-1}(a)\right)$$

Therefore, when we want both $\mathrm{S_{RD}}$ and $\mathrm{S_{RD}}^{-1}$, we need to implement only g, f and f^{-1}. Although we will stick with the *LUT* implementation of S-Box for better performance.

3 Encryption

Encryption with *AES* can be programmed by simply implementing the different steps provided in FIPS 197 documentation[NIS01b].

3.1 SubBytes

The implementation of `SubBytes` requires a table of 256 bytes to store $\mathrm{S_{RD}}$. Which can be simply done as:

```
void SubBytes(uint8_t* state) {
// Replace each byte with its corresponding value from the S-box
    for(int i = 0; i < 16; ++i)
        state[i] = sbox[state[i]];
}
```

This can be optimized with loop unrolling which eliminates the loop control overhead and potentially allowing for more efficient use of the CPU's instruction pipeline. And by accessing elements sequentially, we may take better advantage of spatial locality, reducing cache misses. It also helps modern CPUs prefetch the next data, improving cache performance.
From now on, we would try to use the loop unrolling wherever possible and beneficial.

3.2 ShiftRows

The implementation of `ShiftRows` is straightforward from the description.

3.3 MixColumns

In choosing the `MixColumns` polynomial, we took into account the efficiency on 8 -bit processors. We illustrate how `MixColumns` can be realized in a small series of instructions. For an input column $[ABCD]^T$

$$\begin{aligned} Z &= \{01\}A \oplus \{01\}B \oplus \{02\}C \oplus \{03\}D \\ &= A \oplus B \oplus \{02\}C \oplus C \oplus C \oplus \{02\}D \oplus D \\ &= (A \oplus B \oplus C \oplus D) \oplus D \oplus \{02\}(C \oplus D) \end{aligned}$$

The listing gives the algorithm to process one column. The only finite field multiplication used in this algorithm is multiplication with the element 02 , denoted by `xtime`.

$$\begin{array}{lll} t = a[0] \oplus a[1] \oplus a[2] \oplus a[3]; & \text{a is a column} & \\ u = a[0]; & & \\ v = a[0] \oplus a[1]; & v = \text{xtime}(v); & a[0] = a[0] \oplus v \oplus t \\ v = a[1] \oplus a[2]; & v = \text{xtime}(v); & a[1] = a[1] \oplus v \oplus t \\ v = a[2] \oplus a[3]; & v = \text{xtime}(v); & a[2] = a[2] \oplus v \oplus t \\ v = a[3] \oplus u; & v = \text{xtime}(v); & a[3] = a[3] \oplus v \oplus t \end{array}$$

3.4 AddRoundKey

To make the `AddRoundKey` more efficient, we can leverage vectorization techniques, assuming the target platform supports **SIMD** (Single Instruction Multiple Data) instructions. This allows us to process multiple bytes in parallel, significantly speeding up the operation.

3.5 KeyExpansion

Since KeyExpansion is invoked only once per run, we didn't focus on optimization and simply followed the description.

4 Decryption

For implementations on 8-bit platforms, there is no benefit in following the equivalent decryption algorithm. Instead, the straightforward decryption algorithm is followed. Decryption is similar in structure to encryption, but uses the `InvMixColumns` step instead of `MixColumns`. Where the `MixColumns` coefficients are limited to 01, 02 and 03, the coefficients of `InvMixColumns` are 09, OE, OB and OD. In our implementation, these multiplications take significantly more time and this results in a small performance degradation. A considerable speed-up can be obtained by using look-up tables at the cost of additional tables.
P. Barreto observed the following relation between the `MixColumns` polynomial $c(x)$ and the `InvMixColumns` polynomial $d(x)$:

$$d(x) = \left(04x^2 + 05\right) c(x) \quad \bmod \left(x^4 + 01\right) \tag{1}$$

In matrix notation, this relation becomes:

$$\begin{bmatrix} \text{OE} & \text{OB} & \text{OD} & \text{O9} \\ \text{O9} & \text{OE} & \text{OB} & \text{OD} \\ \text{OD} & \text{O9} & \text{OE} & \text{OB} \\ \text{OB} & \text{OD} & \text{O9} & \text{OE} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} 05 & 00 & 04 & 00 \\ 00 & 05 & 00 & 04 \\ 04 & 00 & 05 & 00 \\ 00 & 04 & 00 & 05 \end{bmatrix}$$

The consequence is that `InvMixColumns` can be implemented as a simple preprocessing step, followed by a `MixColumns` step. An algorithm for the preprocessing step is given below. If the small performance drop caused by this implementation of the preprocessing step is acceptable, no extra tables have to be defined.

$$\begin{aligned}
&u = \text{xtime}(\text{xtime}(a[0] \oplus a[2])); \text{a is a column}\\
&v = \text{xtime}(\text{xtime}(a[1] \oplus a[3]));\\
&a[0] = a[0] \oplus u;\\
&a[1] = a[1] \oplus v;\\
&a[2] = a[2] \oplus u;\\
&a[3] = a[3] \oplus v;
\end{aligned}$$

5 Techniques for Implementing AES: An Overview

5.1 Look Up Tables

The different steps of the round transformation can be combined in a single set of look-up tables, allowing for very fast implementations on processors with word lengths 32 or greater. In this section, we explain how this can be done.

Let the input of the round transformation be denoted by **a**, and the output of `SubBytes` by **b**:

$$b_{i,j} = \mathrm{S_{RD}}\left[a_{i,j}\right], \quad 0 \le i < 4; 0 \le j < \mathrm{N_b} \tag{1}$$

Let the output of `ShiftRows` be denoted by **c** and the output of `MixColumns` by **d** :

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j+C_0} \\ b_{1,j+C_1} \\ b_{2,j+C_2} \\ b_{3,j+C_3} \end{bmatrix}, 0 \le j < \mathrm{N_b} \tag{2}$$

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}, 0 \le j < \mathrm{N_b} \tag{3}$$

The addition in the indices of (2) must be done in modulo $\mathrm{N_b}$. Equations (1) – (3) can be combined into:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}, 0 \le j < \mathrm{N_b} \tag{4}$$

The matrix multiplication can be interpreted as a linear combination of four column vectors:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \mathrm{S_{RD}}\left[a_{0,j+C_0}\right] \oplus \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \mathrm{S_{RD}}\left[a_{1,j+C_1}\right] \oplus \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \mathrm{S_{RD}}\left[a_{2,j+C_2}\right] \oplus \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \mathrm{S_{RD}}\left[a_{3,j+C_3}\right] \tag{5}$$

We define now the four T-tables: T_0, T_1, T_2 and T_3 :

$$T_0[a] = \begin{bmatrix} 02 \cdot \mathrm{S_{RD}}[a] \\ 01 \cdot \mathrm{S_{RD}}[a] \\ 01 \cdot \mathrm{S_{RD}}[a] \\ 03 \cdot \mathrm{S_{RD}}[a] \end{bmatrix}, T_1[a] = \begin{bmatrix} 03 \cdot \mathrm{S_{RD}}[a] \\ 02 \cdot \mathrm{S_{RD}}[a] \\ 01 \cdot \mathrm{S_{RD}}[a] \\ 01 \cdot \mathrm{S_{RD}}[a] \end{bmatrix}, T_2[a] = \begin{bmatrix} 01 \cdot \mathrm{S_{RD}}[a] \\ 03 \cdot \mathrm{S_{RD}}[a] \\ 02 \cdot \mathrm{S_{RD}}[a] \\ 01 \cdot \mathrm{S_{RD}}[a] \end{bmatrix}, T_3[a] = \begin{bmatrix} 01 \cdot \mathrm{S_{RD}}[a] \\ 01 \cdot \mathrm{S_{RD}}[a] \\ 03 \cdot \mathrm{S_{RD}}[a] \\ 02 \cdot \mathrm{S_{RD}}[a] \end{bmatrix}$$

These tables have each 256 4-byte word entries and require 4 kB of storage space. Using these tables, (5) translates into:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = T_0\left[a_{0,j+C_0}\right] \oplus T_1\left[a_{1,j+C_1}\right] \oplus T_2\left[a_{2,j+C_2}\right] \oplus T_3\left[a_{3,j+C_3}\right], 0 \le j < \mathrm{N_b}$$

Taking into account that `AddRoundKey` can be implemented with an additional 32-bit XOR operation per column, we get a look-up table implementation with 4 kB of tables that takes only four table look-ups and four XOR operations per column per round.For simple blocs, lookup tables are fast but they are sensible to timings attacks.

5.2 With Inversion in $GF(2^8)$

The problem of designing efficient circuits for inversion in finite fields has been studied extensively before; e.g. by C. Paar and M. Rosner[PR97].This problem of generating an inverse in GF $\left(2^8\right)$ has been translated into the calculation of an inverse and some operations in GF $\left(2^4\right)$. The calculation of an inverse in GF $\left(2^4\right)$ can be done with a small table. A. Satoh et al.[Sat+01] showed that the hardware size of S-Box using the field GF$(((2^2)^2)^2)$ is 294 gates, which is about 20% smaller and slightly faster than the one using the field GF$((2^4)^2)$.
Later D. Canright[Can05] prposed a very compact S-Box for AES where best case improved by 20%. This decreased size helps for area-limited hardware implementations, e.g., smart cards, and to allow more copies of the S-box for parallelism and/or pipelining of AES.

5.3 Bit Sliced

In order to gain speed, we can have a look at a bit sliced version. Bit slicing is basically writing an hardware implementation in software, considering that each bit is a different input and operations as gates applied at the same time.
In 2007 Matsui and Nakajima[MN07] propose a bit-sliced implementation of AES-CTR, about 30% faster than the table lookup version. It uses 128 bits vector registers and computes 128 blocs of AES at the same time. The only draw back is that you need 2kB of data to encrypt to benefit from the speed up. But that is still useful in the case of HDD encryption.
Two years later P. Schwabe and E. Käsper[KS09] implement another version 20% faster and that does not have this 2kB requirement.

5.4 But I need more SPEED !

In 2010, Intel provides a hardware AES instruction set[Mob10].

```
aesenc ,  aesenclast , aesdec ,  aesdeclast , keygenassist
```

Which leads to the following:

```
# The data block is in xmm15 and Registers xmm0–xmm10 hold the round keys
pxor xmm15, xmm0     ; Whitening step (Round 0)
aesenc xmm15, xmm1   ; Round 1
aesenc xmm15, xmm2   ; Round 2
aesenc xmm15, xmm3   ; Round 3
aesenc xmm15, xmm4   ; Round 4
aesenc xmm15, xmm5   ; Round 5
aesenc xmm15, xmm6   ; Round 6
aesenc xmm15, xmm7   ; Round 7
aesenc xmm15, xmm8   ; Round 8
aesenc xmm15, xmm9   ; Round 9
aesenclast xmm15, xmm10       ; Round 10
```

Using AES-NI eliminates the need for AES lookup tables, which have been a source of cache-related timing side channel vulnerabilities[OST10].

6 Modes of Operation for Confidentiality

Here we have discussed only those modes of operations which were mentioned by nist[NIS01a].
Among all the modes the input to the encryption processes of the CBC, CFB, and OFB modes includes, in addition to the plaintext, a data block called the initialization vector, denoted IV. The IV is used in an initial step in the encryption of a message and in the corresponding decryption of the message. Here we have used **openssl** rand to generate random IV.

For the ECB, CBC the plaintext must be a sequence of one or more complete data blocks . In order to make the plaintext length multiple of number of bytes in a block, we have used PKCS#7 padding[Kal98].

6.1 ECB Mode

This mode corresponds to the naive use of a block cipher: given a sequence $m_0\ m_1\ \ldots$ of plaintext blocks, each m_i is encrypted with the same key K, producing a string of ciphertext blocks, $c_0\ c_1\ \ldots$ (after padding with PKCS#7) ECB mode is virtually never used in practice. One obvious weakness of ECB mode is that the encryption of identical plaintext blocks yields identical ciphertext blocks. This is a serious weakness if the underlying message blocks are chosen from a "low entropy" plaintext space. To take an extreme example, if a plaintext block always consists entirely of 0's or entirely of 1's, then ECB mode is essentially useless.

6.2 CBC Mode

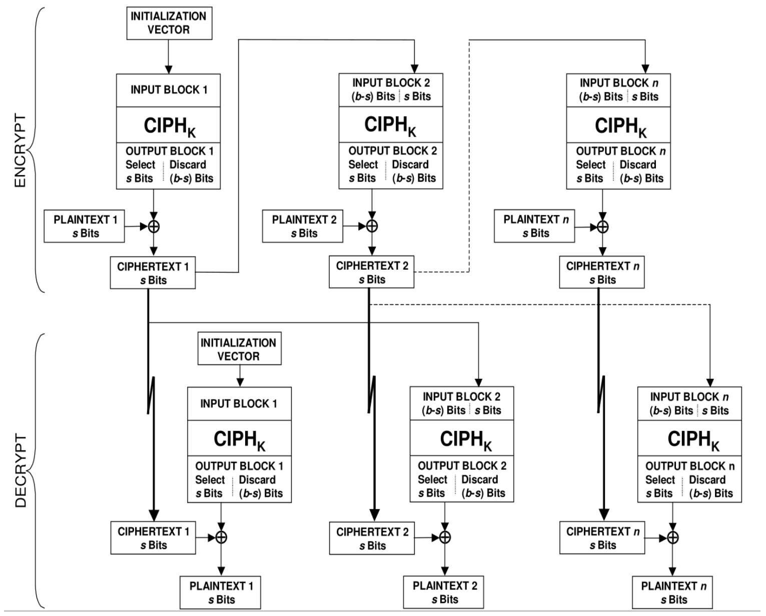
In CBC mode, each ciphertext block c_i is x-ored with the next plaintext block, m_{i+1} , before being encrypted with the key K. More formally, we start with an initialization vector, denoted by IV (Note that IV has the same length as a plaintext block). Then we construct c_0 , c_1 , ..., using the rule

$$Encryption \begin{cases} c_0 = E_K(m_0 \oplus IV) \\ c_i = E_K(m_i \oplus c_{i-1}), \quad i \geq 1; \end{cases} \qquad Decryption \begin{cases} m_0 = D_K(c_0) \oplus IV \\ m_i = D_K(c_i) \oplus (c_{i-1}), \quad i \geq 1; \end{cases}$$

Although the random IV makes CBC mode CPA secure, in order to provide protection from CCA attacks in this mode of operation, it is necessary to use Authenticated Encryption.

6.3 128-bit CFB Mode

CFB mode features the feedback of successive ciphertext segments into the input blocks of the forward cipher to generate output blocks that are exclusive-ORed with the plaintext to produce the ciphertex. We start with IV (an initialization vector) and we produce the keystream element z_i by encrypting the previous encryption block output. That is,


ENCRYPT
DECRYPT
INITIALIZATION VECTOR
INPUT BLOCK 1
INPUT BLOCK 2 (b-s) Bits | s Bits
INPUT BLOCK n (b-s) Bits | s Bits
CIPH_K
OUTPUT BLOCK 1 Select s Bits | Discard (b-s) Bits
OUTPUT BLOCK 2 Select s Bits | Discard (b-s) Bits
OUTPUT BLOCK n Select s Bits | Discard (b-s) Bits
PLAINTEXT 1 s Bits
PLAINTEXT 2 s Bits
PLAINTEXT n s Bits
CIPHERTEXT 1 s Bits
CIPHERTEXT 2 s Bits
CIPHERTEXT n s Bits

$$Encryption \begin{cases} z_0 = E_K(IV) \\ c_i = E_K(z_i) \oplus m_i \\ z_{i+1} = c_i, \quad i \geq 0; \end{cases} \qquad Decryption \begin{cases} z_0 = E_K(IV) \\ m_i = E_K(z_i) \oplus c_i \\ z_{i+1} = c_i, \quad i \geq 0; \end{cases}$$

Note that the encryption function E_K is used for both encryption and decryption in CFB mode. Like CBC mode, changes in the plaintext propagate forever in the ciphertext, and encryption cannot be parallelized. Also like CBC, decryption can be parallelized.

6.4 OFB Mode

In OFB mode, a keystream is generated, which is then x-ored with the plaintext (i.e., it operates as a stream cipher). OFB mode is actually a synchronous stream cipher: the keystream is produced by repeatedly encrypting an initialization vector, IV. Encryption and decryption are done using the rule

$$Encryption \begin{cases} z_0 = IV \\ z_i = E_K(z_{i-1}) \\ c_i = z_i \oplus m_i, \quad i \geq 1; \end{cases} \qquad Decryption \begin{cases} z_0 = IV \\ z_i = E_K(z_{i-1}) \\ m_i = z_i \oplus c_i, \quad i \geq 1; \end{cases}$$

Again, the encryption function E_K is used for both encryption and decryption in OFB mode. Rogaway[Uni11] pointed out that the OFB does not offer security from CCA attacks.

6.5 CTR Mode

Counter mode is similar to OFB mode; the only difference is in how the keystream is constructed. Suppose that the length of a block in bits is denoted by l. In counter mode, we choose a counter, denoted ctr, which is a bitstring of length l. Then we construct a sequence of bitstrings of length l, denoted T_1 , T_2 , . . . , defined as follows:

$$Encryption \begin{cases} Ti = (ctr + i{-}1) mod\ 2^l \\ c_i = E_K(T_i) \oplus m_i, \quad i \geq 1; \end{cases} \qquad Decryption \begin{cases} Ti = (ctr + i{-}1) mod\ 2^l \\ m_i = E_K(T_i) \oplus c_i, \quad i \geq 1; \end{cases}$$

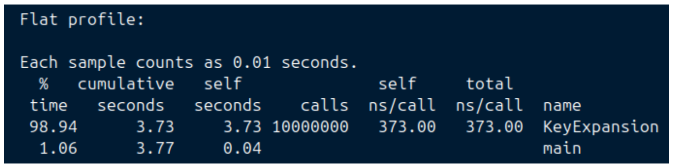
Usually the counter is initialized to some value, and then it is incremented by one for every block. Simply adding or XORing the nonce and counter into a single value would break the security under a chosen-plaintext attack in many cases, since the attacker may be able to manipulate the entire IV–counter pair to cause a collision. In his book C. Paar[PR97] explained that the initializing value of the counter is a non-repeatable number on the order of 96 bits. The other 32 bits are zero at the beginning of the process, and then their values are incremented by 1 for every block.
In summation on the modes of operation in, the CTR mode is marked as the best choice among all the others, as it provides software & hardware efficiency, pre-processing, parallelization, local error(useful for error-code correction).

7 Performance Analysis

The experiments were carried out on a AMD RYZEN3 processor based on ZEN+ microarchitecture codename PICASSO running at 2.23 GHz. The system was run without any kind of frequency boost, multi-threading or multiprocessing. Any kind of GCC optimization was not used. The operating system was Linux (Ubuntu 24.04 64 bits).
All the profiling has been done using **GPROF** and `clock()` function.

7.1 Key Expansion

For AES-128 key expansion, it took an average of $(3.73 \times 2.23\times 10^9 \times/10^7)$= 832 cycles



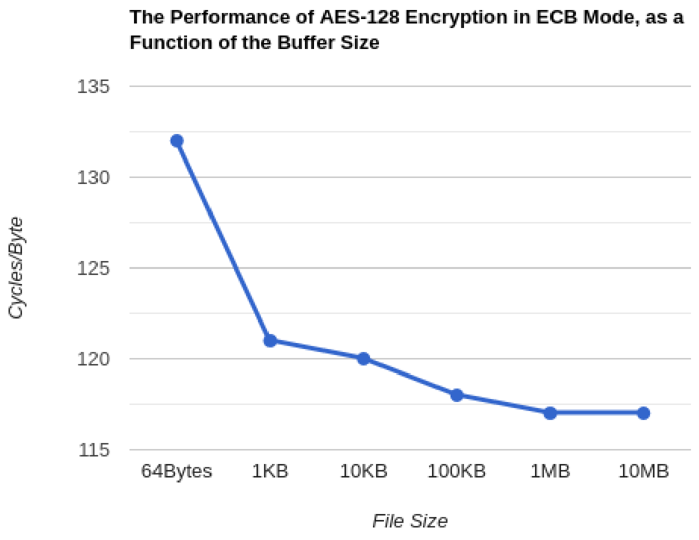
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 98.94      3.73     3.73 10000000   373.00   373.00  KeyExpansion
  1.06      3.77     0.04                             main
```

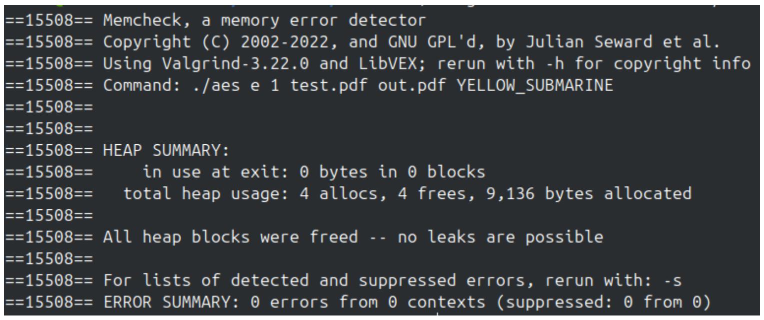
7.2 Encryption and Decryption

Encryption of a single block took an average of 90 cycles/byte.
while decryption of a single block took an average of 115 cycles/byte.

Performance in CPU Cycles Per Byte for a 1mb buffer					
	ECB mode	CBC mode	CFB mode	OFB mode	CTR mode
Encryption	117	123	122	123	121
Decryption	179	196	124	128	122



Our implementation was able to produce a throughput of about 34MB/s while the look up table version had throughput of 400MB/s and the AES-NI implementation achieved a massive throughput of approximately 1.3GB/s. For any potential memory leak, we have used **valgrind** to check that. Here is the summary report:



```
==15508== Memcheck, a memory error detector
==15508== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==15508== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==15508== Command: ./aes e 1 test.pdf out.pdf YELLOW_SUBMARINE
==15508==
==15508==
==15508== HEAP SUMMARY:
==15508==     in use at exit: 0 bytes in 0 blocks
==15508==   total heap usage: 4 allocs, 4 frees, 9,136 bytes allocated
==15508==
==15508== All heap blocks were freed -- no leaks are possible
==15508==
==15508== For lists of detected and suppressed errors, rerun with: -s
==15508== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

References

[PR97] Christof Paar and Martin Rosner. "Comparison of Arithmetic Architectures for Reed-Solomon Decoders in Reconfigurable Hardware". In: *The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186, 7th International Workshop, Napa Valley, CA, USA , 16-18 April 1997, Proceedings.* Lecture Notes in Computer Science. IEEE, 1997. DOI: 10.1109/fpga.1997.624622. URL: https://sci-hub.se/10.1109/fpga.1997.624622.

[NIS01a] NIST. "Recommendation for block cipher modes of operation". In: *Technical Report SP 800-38A* (2001). URL: https://doi.org/10.6028/NIST.SP.800-38A.

[NIS01b] NIST. "Specification for the ADVANCED ENCRYPTION STANDARD (AES)". In: *Technical Report FIPS PUB 197* (2001). URL: https://doi.org/10.6028/NIST.FIPS.197-upd1.

[Sat+01] Akashi Satoh et al. "A Compact Rijndael Hardware Architecture with S-Box Optimization". In: *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings.* Vol. 2248. Lecture Notes in Computer Science. Springer, 2001, pp. 239–254. DOI: 10.1007/3-540-45682-1_15. URL: https://iacr.org/archive/asiacrypt2001/22480241.pdf.

[Can05] David Canright. "A Very Compact S-Box for AES". In: *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings.* Vol. 3659. Lecture Notes in Computer Science. Springer, 2005, pp. 441–455. DOI: 10.1007/11545262_32. URL: https://iacr.org/archive/ches2005/032.pdf.

[MN07] Mitsuru Matsui and Junko Nakajima. "On the Power of Bitslice Implementation on Intel Core2 Processor". In: *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings.* Vol. 4727. Lecture Notes in Computer Science. Springer, 2007, pp. 121–134. DOI: 10.1007/978-3-540-74735-2_9. URL: https://iacr.org/archive/ches2007/47270121/47270121.pdf.

[KS09] Emilia Käsper and Peter Schwabe. "Faster and Timing-Attack Resistant AES-GCM". In: *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings.* Vol. 5747. Lecture Notes in Computer Science. Springer, 2009, pp. 1–17. DOI: 10.1007/978-3-642-04138-9_1. URL: https://www.iacr.org/archive/ches2009/57470001/57470001.pdf.

[OST10] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Efficient Cache Attacks on AES, and Countermeasures". In: *J. Cryptology* 23 (2010), pp. 37–71. DOI: 10.1007/s00145-009-9049-y. URL: https://cs-people.bu.edu/tromer/papers/cache-joc-20090619.pdf.

[Joa01] Vincent Rijmen Joan Daemen. "The Design of Rijndael AES — The Advanced Encryption Standard". In: *Springer-Verlag* (November26, 2001). URL: https://cs.ru.nl/~joan/papers/JDA_VRI_Rijndael_2002.pdf.

[Kal98] B. Kaliski. *Request for Comments: 2315.* Tech. rep. RSA Laboratories, East, March 1998. URL: https://datatracker.ietf.org/doc/html/rfc2315.

[Mob10] Shay Gueron Mobility Group Israel Development Center Intel Corporation. *White Paper: Intel Advanced Encryption Standard (AES) New Instructions Set.* May 2010. URL: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf.

[Uni11] Phillip Rogaway University of California. *Evaluation of Some Blockcipher Modes of Operation.* February 10, 2011. URL: https://www.cs.ucdavis.edu/~rogaway/papers/modes.pdf.