AES-128 Implementation using Intel's AES-NI instructions: Documentation

Tanushri Sen
Mtech CrS2324

September 16, 2024

Contents

1 Introduction

This document presents the implementation of AES encryption and decryption using Intel's AES-NI instruction set. AES (Advanced Encryption Standard) is a symmetric encryption algorithm widely used for securing data. The program performs key expansion, encryption, and decryption using AES-NI intrinsics for efficient hardware-level encryption.

1.1 What is AES-NI instructions?

The new AES-NI instruction set is comprised of six new instructions that perform several compute intensive parts of the AES algorithm. These instructions can execute using significantly less clock cycles than a software solution. Four of the new instructions are for accelerating the encryption/decryption of a round and two new instructions are for round key generation. The following is a description of the new instructions.

- **AESENC.** This instruction performs a single round of encryption. The instruction combines the four steps of the AES algorithm - `ShiftRows`, `SubBytes`, `MixColumns` and `AddRoundKey` into a single instruction.
- **AESENCLAST.** Instruction for the last round of encryption. Combines the `ShiftRows`, `SubBytes`, and `AddRoundKey` steps into one instruction.
- **AESDEC.** Instruction for a single round of decryption. This combines the four steps of AES - `InvShiftRows`, `InvSubBytes`, `InvMixColumns`, `AddRoundKey` into a single instruction.
- **AESDECLAST.** Performs last round of decryption. It combines `InvShiftRows`, `InvSubBytes`, `AddRoundKey` into one instruction.
- **AESKEYGENASSIST** is used for generating the round keys used for encryption.
- **AESIMC.** is used for converting the encryption round keys to a form usable for decryption using the Equivalent Inverse Cipher.

1.2 Overview of the Implementation

This document provides an overview of an AES-128 implementation using Intel AES-NI intrinsics, which accelerate cryptographic operations. Main components for discussion:

- **KeyExpansion():** Generates the 10 round keys for AES-128 from the initial key using the `AESkeygenassist` intrinsic. Each round key is derived by applying XOR and shifts to the previous key.
- **Encrypt():** Encrypts a 128-bit plaintext using AES. It performs an initial key addition, followed by 9 rounds of encryption (`AESENC`), and a final round (`AESENCLAST`).
- **Decrypt():** Decrypts a 128-bit ciphertext by applying the round keys in reverse. It inverts the round keys using `AESIMC` and uses the `AESDEC` instruction for deryption.

The main function demonstrates encryption and decryption of a sample plaintext.

2 Function Documentation

2.1 KeyExpansion()

It takes a key and produces the key schedule used for encryption and decryption.

```
1 void KeyExpansion(unsigned char *key, __m128i *key_schedule);
```

The algorithm and logic used for above function are given below:

KeyExpansion(key, key_schedule)

```
begin
    key_schedule[0] = [key_3 : key_2 : key_1 : key_0] each key_i is of 32-bits

    for i = 1 to 10 (stepsize 1):
        switch (i):
            case 1: A = AESKEYGENASSIST(key_schedule[i-1], 0x01) break
            case 2: A = AESKEYGENASSIST(key_schedule[i-1], 0x02) break
            case 3: A = AESKEYGENASSIST(key_schedule[i-1], 0x04) break
            case 4: A = AESKEYGENASSIST(key_schedule[i-1], 0x08) break
            case 5: A = AESKEYGENASSIST(key_schedule[i-1], 0x10) break
            case 6: A = AESKEYGENASSIST(key_schedule[i-1], 0x20) break
            case 7: A = AESKEYGENASSIST(key_schedule[i-1], 0x40) break
            case 8: A = AESKEYGENASSIST(key_schedule[i-1], 0x80) break
            case 9: A = AESKEYGENASSIST(key_schedule[i-1], 0x1B) break
            case 10: A = AESKEYGENASSIST(key_schedule[i-1], 0x36) break
            default: return // Invalid round index

        // A = [A_3 : A_2 : A_1 : A_0]
        B = [A_3 : A_3 : A_3 : A_3]

        B = B XOR (key_schedule[i-1] << 4 bytes)
        B = B XOR (key_schedule[i-1] << 8 bytes)
        B = B XOR (key_schedule[i-1] << 12 bytes)

        key_schedule[i] = B XOR key_schedule[i-1]
    end for
end
```

Description of above algorithm:

- **Initialization:** Store given 16 bytes initial key into a 128-bit register (`__m128i`) `key_schedule[0]`, where Each entry in the key schedule consists of four 32-bit words.
- **Round Key Generation:** For each round (from 1 to 10), the round constant is used to generate a new round key. The `AESKEYGENASSIST` intrinsic is employed to assist in this process. The round constant varies for each round, as defined in the following array:

  `Rcon[] = {0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36}`

  and in `i`-th round `Rcon[i]` is used. Now, the output of `AESKEYGENASSIST` is stored into another `__m128i` register, `A` as `[A_3 : A_2 : A_1 : A_0]`.
- **Transformation:** The result from `AESKEYGENASSIST` is used to generate a temporary value `B`. The most significant 32 bits of `A` are replicated across all four 32-bit chunks of `B`. `B` is then XOR-ed with shifted versions of `key_schedule[i-1]` to compute the new round key.
- **Finalization:** The new round key is obtained by XOR-ing `B` with `key_schedule[i-1]`, and it is stored in `key_schedule[i]`.

This process ensures that each round key is uniquely generated and prepared for use in AES encryption and decryption.

2.2 Encrypt()

This function encrypts the plaintext using the provided key schedule and outputs the ciphertext.

```
void Encrypt(const unsigned char *plaintext, unsigned char *cipher, const __m128i
    *key_schedule);
```

The algorithm used to encrypt given 16 bytes plaintext as follows:

Encrypt(plaintext, cipher, key_schedule)

```
begin
    block = plaintext

    block = block XOR key_schedule[0]

    for i = 1 to 9 (step size 1)
        block = AESENC(block, key_schedule[i])
    end for
    block = AESENCLAST(block, key_schedule[10])

    cipher = block
end
```

Description of above algorithm:

- **Load Plaintext:** The plaintext is loaded into a `__m128i block` using the intrinsic `_mm_loadu_si128`.
- **Initial Round Key Addition:** The initial round key is applied to the block using the XOR operation.
- **AES Rounds:** The function performs 9 rounds of AES encryption using the `AESENC` intrinsic. Each `AESENC` involves substituting bytes, shifting rows, and mixing columns internally.
- **Final Round:** The final round is performed using the `AESENCLAST` intrinsic, which excludes the MixColumns step.
- **Store Encrypted Block:** The encrypted block is stored back into the output cipher.

2.3 Decrypt()

This function decrypts the ciphertext using the key schedule and outputs the plaintext.

```
void Decrypt(const unsigned char *cipher, unsigned char *decipher, const __m128i *
    key_schedule);
```

Algorithm used for decryption given 16 bytes of ciphertext:

Decrypt(cipher, decipher, key_schedule)

```
begin
    block = cipher
    block = block XOR key_schedule[10]
    for i = 9 downto 1 (step size -1)
        block = AESDEC(block, AESIMC(key_schedule[i]))
    end for
    block = AESDECLAST(block, key_schedule[0])

    decipher = block
end
```

Description of above algorithm:

- **Load Ciphertext:** The ciphertext is loaded into a `__m128i block` using the intrinsic `_mm_loadu_si128`.
- **Add Last Round Key:** The last round key is XORed to the block .
- **AES Decryption Rounds:** The 9 rounds decryption is performed as follows:
  - The function `AESIMC(key_schedule[i])` generates an intermediate round key by applying the Inverse MixColumns transformation to the round key `key_schedule[i]`. This transformation prepares the key for use in the AES decryption process.
  - The `AESDEC` operation then decrypts the block using this transformed round key.
- **Final Round:** The final round is performed using the `AESDECLAST` intrinsic, which excludes the Inverse MixColumns step.
- **Store Decrypted Block:** The decrypted block is stored back into the output decipher.

3 Example Usage

Below is an example of how the AES implementation can be used:

```
int main()
{
    // 128-bit key (16 bytes)
    unsigned char key[4 * Nk] = {0x2b, 0x7e, 0x15, 0x16,
                                 0x28, 0xae, 0xd2, 0xa6,
                                 0xab, 0xf7, 0x15, 0x88,
                                 0x09, 0xcf, 0x4f, 0x3c
                                };

    // register to hold the round keys (Nr+1 keys)
    __m128i key_schedule[Nr + 1];
    // Generate the key schedule
    KeyExpansion(key, key_schedule);

    // 128-bit plaintext block (16 bytes)
    unsigned char plaintext[4 * Nb] = {0x32, 0x43, 0xf6, 0xa8,
                                       0x88, 0x5a, 0x30, 0x8d,
                                       0x31, 0x31, 0x98, 0xa2,
                                       0xe0, 0x37, 0x07, 0x34
                                        };
    unsigned char cipher[16];
    unsigned char decipher[16];

    // Encrypt the plaintext
    Encrypt(plaintext, cipher, key_schedule);
    printf("Encrypted text: ");
    for (int i = 0; i < 16; i++) {
        printf("%02x ", cipher[i]);
    }
    printf("\n");

    // Decrypt the ciphertext
    Decrypt(cipher, decipher, key_schedule);

    printf("Decrypted text: ");
    for (int i = 0; i < 16; i++) {
        printf("%02x ", decipher[i]);
```

```
    }
    printf("\n");
    return 0;
}
```

4 Testing and Verification

4.1 Test Vectors

To verify the correctness of the implementation, use the following test vectors:

- **Input Message:** `32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34`
- **Key:** `2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c`
- **Expected Cipher:** `39 25 84 1d 02 dc 09 fb dc 11 85 97 19 6a 0b 32`
- **Expected Decrypted Message:** Matches the original input message.

4.2 Output

The output of the program should be compared against known AES test vectors to ensure the implementation is correct.

5 Conclusion

This AES implementation using Intel AES-NI intrinsics demonstrates an efficient approach to encryption and decryption. By leveraging hardware acceleration, the KeyExpansion, Encrypt, and Decrypt functions optimize performance and security. The use of AES-NI instructions ensures fast and reliable AES operations, making this implementation suitable for applications requiring both high-speed encryption and robust cryptographic security.

Furthermore, different modes of operations can be used to encrypt data streams more than 16 bytes.

6 Acknowledgement

I would like to express my sincere gratitude to my course instructor, **Dr. Sabyasachi Karati**, for his invaluable guidance and support throughout this project. The insights and feedback were crucial in shaping the implementation and ensuring its accuracy. I also appreciate the encouragement and resources provided, which greatly contributed to the successful completion of this work. Thank you for your dedication and mentorship.

7 Appendix

7.1 References

1. Intel Advanced Encryption Standard Instructions (AES-NI)
2. Intel Intrinsics Guide
3. Wikipedia: Advanced Encryption Standard.

7.2 Code Listing

For completeness, the code is provided in attached link: click here