T.C.
FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

BLM442E Computer System Security
Homework Report

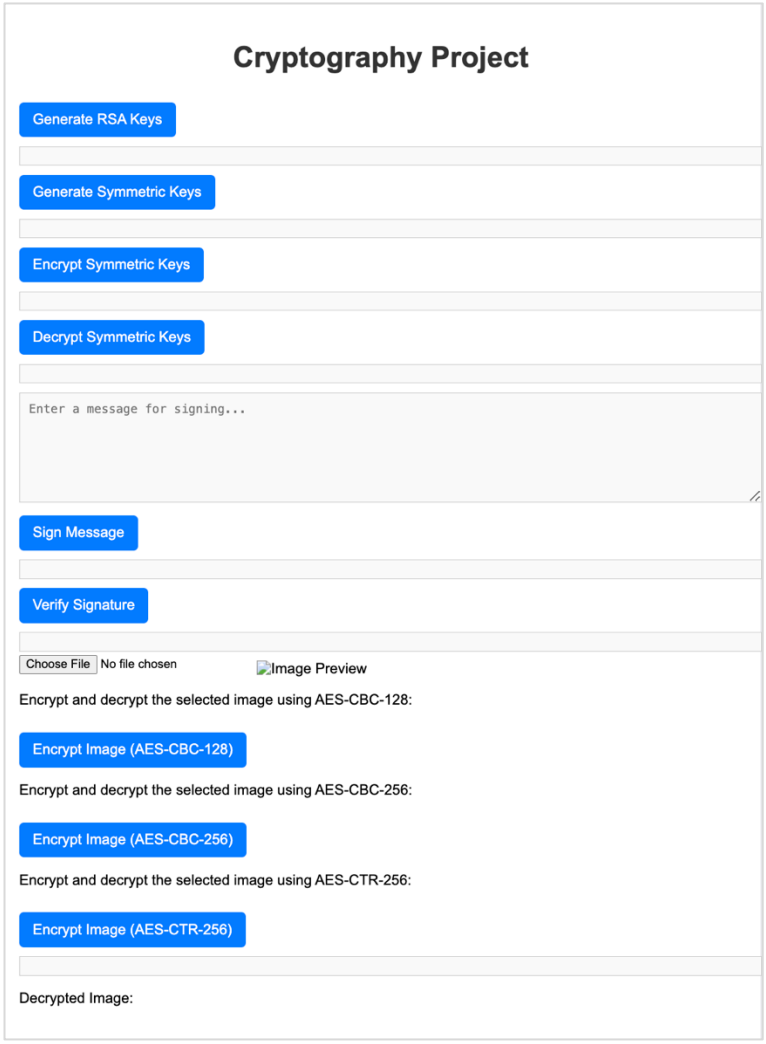
Nagkichan MOUSTAFA

Lecturer : Dr. Ömer KORÇAK

About

This report covers several different encryption algorithms as part of the BLM442E Computer System Security assignment. Tasks include creating RSA key pairs, symmetric keys, applying a SHA-256 hash, signing messages, and encrypting/decrypting an image file using AES in different modes. Details of each task are given below, along with code and results.

I created a project using HTML and JavaScript. The interface is built with HTML, while the cryptology functionality is implemented in JavaScript. The project features a simple design, consisting of buttons and text elements.
When the project is first run, the initial screen appears as shown below. The screen is minimalistic and user-friendly, allowing users to easily navigate and perform the desired operations.
Upon pressing the relevant buttons, the associated operations are executed. Below, I provide a detailed explanation of the functionalities and operations available in the project.



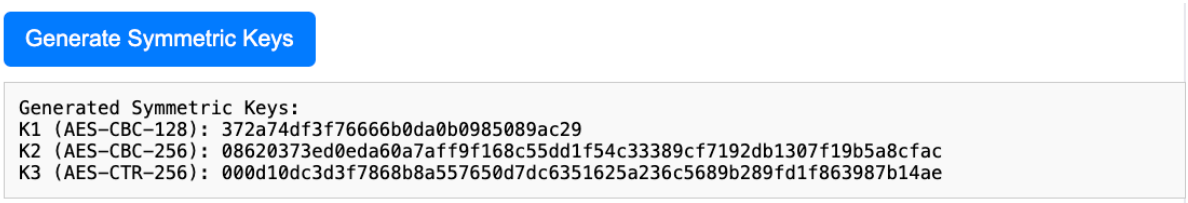
RSA

1. **Generate RSA Keys Button**: A button labeled "Generate RSA Keys" allows the user to generate new RSA keys. When this button is pressed, the cryptographic functions are executed.
2. **RSA Keys Display**: After pressing the button, four sections are populated with the generated keys:
    - **RSA-PSS Public Key**: Displays the RSA-PSS public key.
    - **RSA-PSS Private Key**: Displays the RSA-PSS private key.
    - **RSA-OAEP Public Key**: Displays the RSA-OAEP public key.
    - **RSA-OAEP Private Key**: Displays the RSA-OAEP private key.

Symmetric Keys

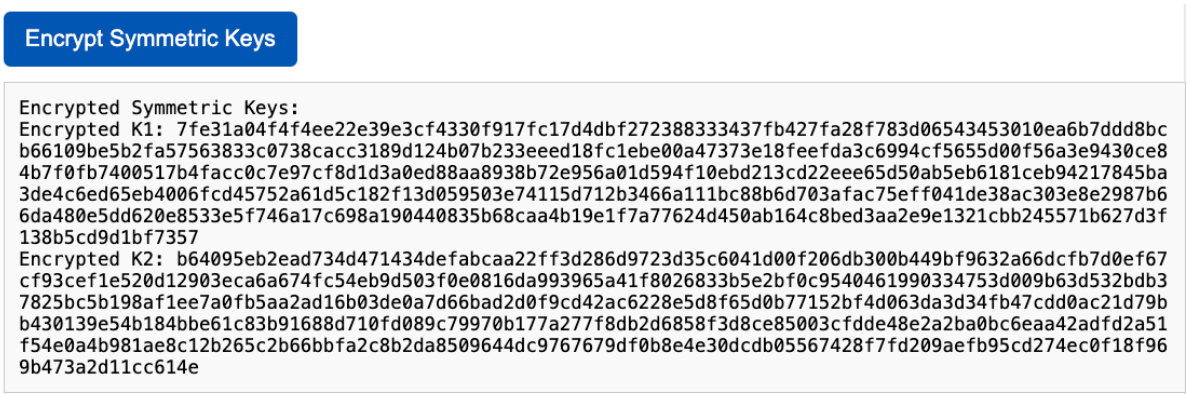
I created three symmetric keys using the **generateSymmetricKeys** function. When click on the button 'Generate Symmetric Keys' generates 3 keys. These keys are as follows:

1. **K1**: 128-bit AES-CBC
2. **K2**: 256-bit AES-CBC
3. **K3**: 256-bit AES-CTR

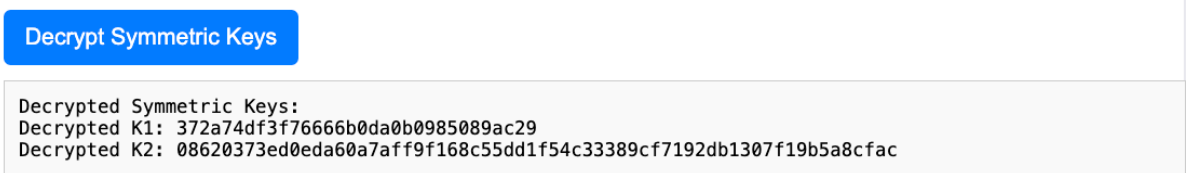


The 'Encypt Symmetric Keys' button performs encryption. Then, to decrypt k1 and k2, I decrypt k1 and k2 by pressing the decrypt symmetric keys button. I performed encryption and decryption operations on these keys as follows:

1. **Encryption**:

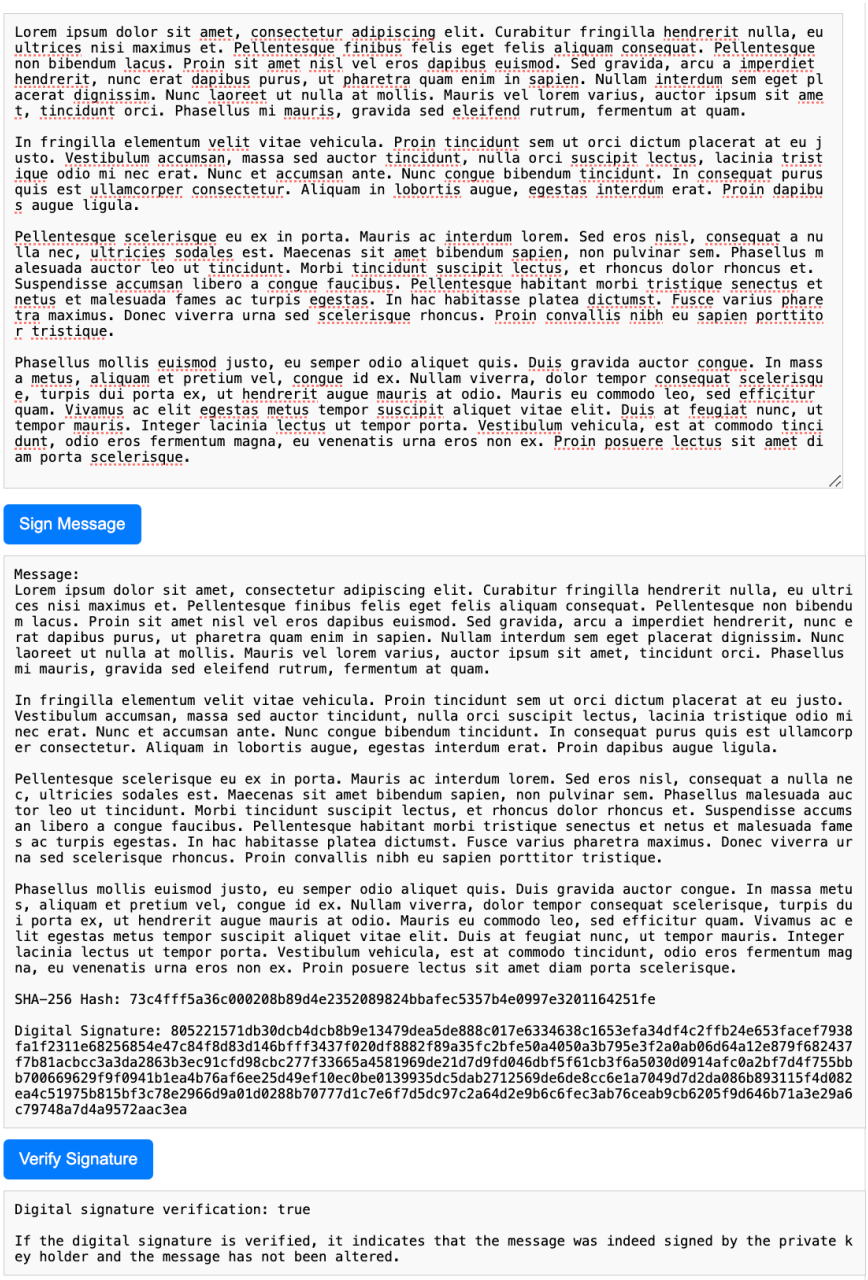


2. **Decryption**:



Message

First of all, I get the message. The text appearing at the top is my message. Then when I press the 'Sign Message' button, the following steps occur:



1) **Hash the Message**: The message is processed using the SHA256 hash algorithm to produce a message digest. This digest is a fixed-length representation of the message, ensuring its integrity.

2) **Create the Digital Signature**: The message digest is then encrypted using the private RSA key (KA−) to generate the digital signature. This digital signature is unique to the message and the private key used.

3) **Display the Results**: The original message, its SHA256 hash, and the digital signature are displayed on the screen.

   a) **Hash the Message Again**: The original message is hashed again using the SHA256 algorithm to produce a new message digest.

   b) **Decrypt the Signature**: The digital signature is decrypted using the public RSA key (KA+), recovering the original message digest.

   c) **Compare the Hashes**: The newly computed message digest is compared with the decrypted message digest. If they match, the signature is valid, confirming that the message was not altered and was indeed signed by the owner of the private key.

```
async function signMessage() {
    try {
        let message = document.getElementById('message').value;
        let encoder = new TextEncoder();
        let data = encoder.encode(message);

        messageHash = await window.crypto.subtle.digest("SHA-256", data);

        signature = await window.crypto.subtle.sign({ name: "RSA-PSS", saltLength: 32 }, rsaKeyPairPSS.privateKey, messageHash);

        const hashHex = bufferToHex(new Uint8Array(messageHash));
        const signatureHex = bufferToHex(new Uint8Array(signature));

        document.getElementById('signature-output').innerText = `Message:\n${message}\n\nSHA-256 Hash: ${hashHex}\n\nDigital Signature: ${signatureHex}`;

        saveToServer("hash_and_signature.txt", `Message:\n${message}\n\nSHA-256 Hash: ${hashHex}\n\nDigital Signature: ${signatureHex}`);
    } catch (error) {
        console.error("Error signing message:", error);
    }
}

async function verifySignature() {
    try {
        let message = document.getElementById('message').value;
        let encoder = new TextEncoder();
        let data = encoder.encode(message);
        let hash = await window.crypto.subtle.digest("SHA-256", data);

        let isValid = await window.crypto.subtle.verify({ name: "RSA-PSS", saltLength: 32 }, rsaKeyPairPSS.publicKey, signature, hash);
        document.getElementById('verification-output').innerText = `Digital signature verification: ${isValid}\n\nIf the digital signature is verified, it indi
    } catch (error) {
        console.error("Error verifying signature:", error);
    }
}
```

The digital signature, along with the message and its hash, is saved to a file named **hash_and_signature.txt**. This file is uploaded to the server using the **server.js** script. The relevant part of the **server.js** script handles the file upload and saves it to the server directory.

```
app.post('/save-hash-signature', (req, res) => {
    if (!req.files || Object.keys(req.files).length === 0) {
        return res.status(400).send('No files were uploaded.');
    }

    let hashSignatureFile = req.files.file;
    let savePath = path.join(__dirname, 'hash_and_signature.txt');

    hashSignatureFile.mv(savePath, (err) => {
        if (err) return res.status(500).send(err);
        res.send('Hash and signature file saved!');
    });
});
```

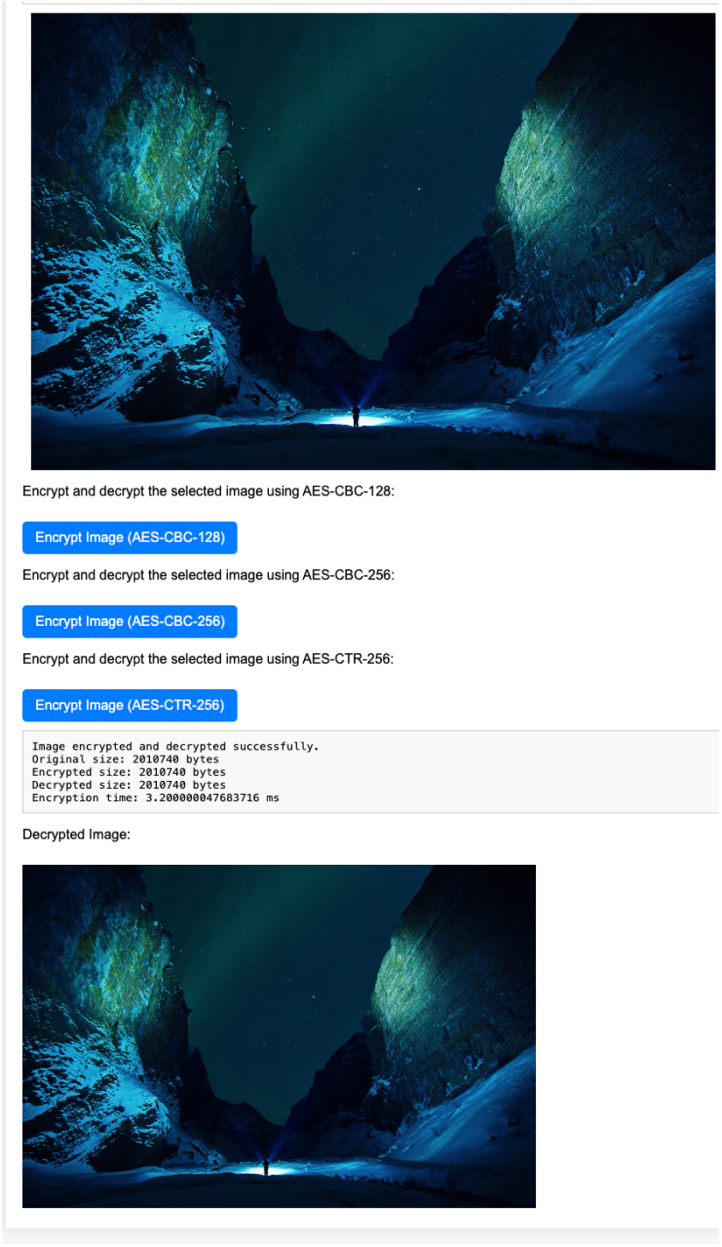
Image Encryption

I used a jpg of approximately 2MB in size to test. First, I select the photo from the files. I added the photo I used to the project file (image.jpg). Then I ran it separately for 3 different modes. I did both encrypt and decrypt in all of them. The photo above is the photo I chose from the files and uploaded. The decrypted photo under the heading 'Decrypted Image:'. Again, I use **server.js** and save it in the project file as **encrypted_image.bin** and **decrypted_image.jpg**

Let's examine the modes one by one:

1. AES (128 bit key) in CBC mode

The picture above is selected by user. When click on the button 'Encrypt Image (AES-CBC-128)' retuns the Image Size and Encryption Time.

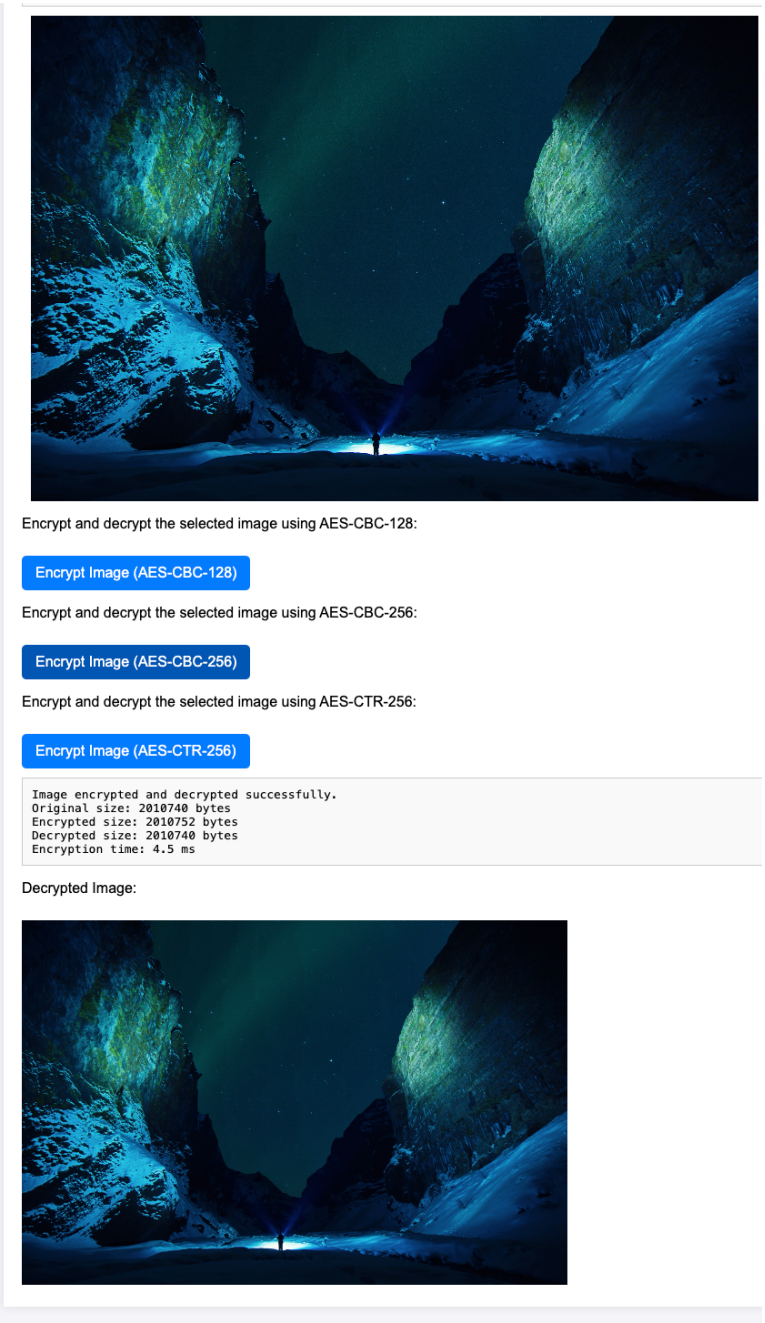


```
Image encrypted and decrypted successfully.
Original size: 2010740 bytes
Encrypted size: 2010740 bytes
Decrypted size: 2010740 bytes
Encryption time: 3.200000047683716 ms
```

Encryption time for CBC 128 bits key is ~ 3. 2000 ms.

And the picture below is the decrypted picture of the selected encrypted picture.

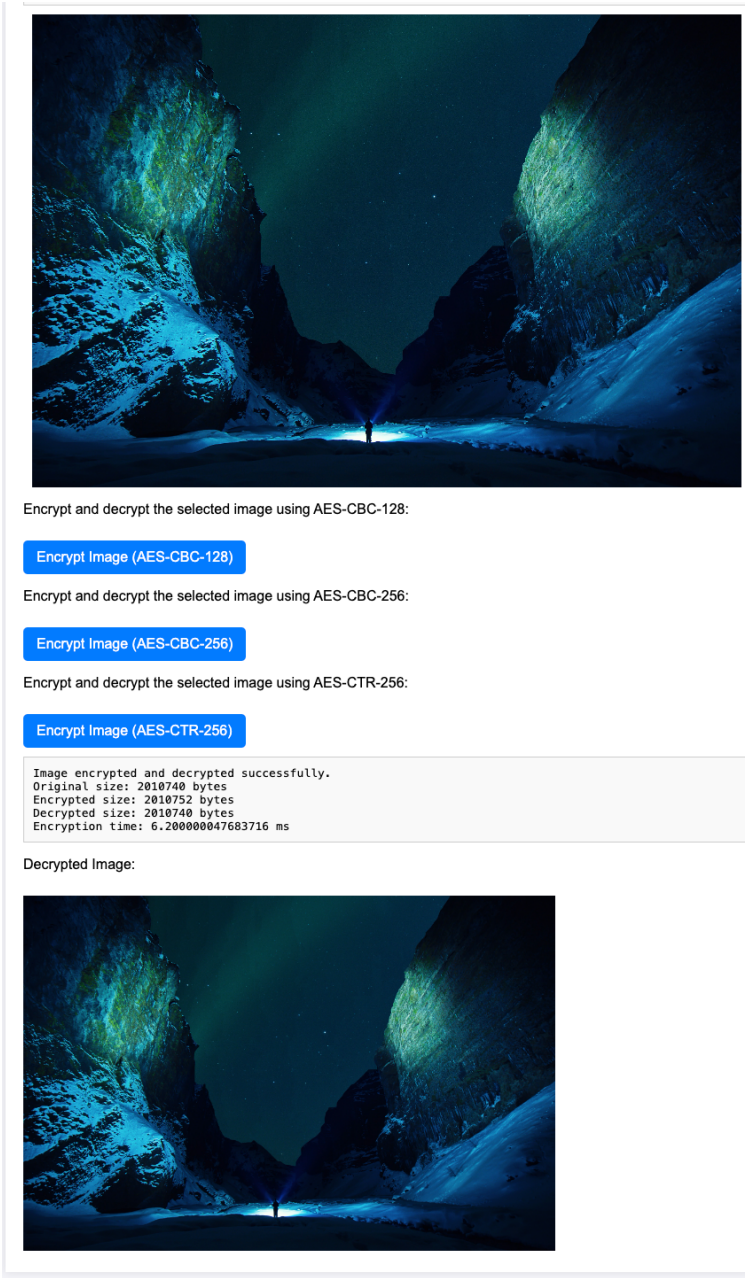
2. AES (256 bit key) in CBC mode.



```
Image encrypted and decrypted successfully.
Original size: 2010740 bytes
Encrypted size: 2010752 bytes
Decrypted size: 2010740 bytes
Encryption time: 4.5 ms
```

Encryption time for CBC 256 bits key is 4. 5 ms.

3. AES (256 bit key) in CTR mode.



```
Image encrypted and decrypted successfully.
Original size: 2010740 bytes
Encrypted size: 2010752 bytes
Decrypted size: 2010740 bytes
Encryption time: 6.200000047683716 ms
```

Encryption time for CTR 256 bits key is ~6.200 ms.

Results for Image Encryption:

Mode (AES)	Bit	Time (ms)
CBC	128 bits	3.20000004768716
CBC	256 bits	4.5
CTR	256 bits	6.20000004768716