SIL765 : Network and Web security

Gaurav Kumar 2021CS10116

February 8, 2024

Abstract

This study evaluates and compares the computational, communication, and storage costs of various cryptographic algorithms used for encryption and authentication purposes. The computational cost is assessed in terms of the time (in milliseconds) required to execute each algorithm, while the communication cost is measured by the length (in bits) of the packet transmitted from the sender to the receiver during the algorithm's operation. Additionally, the storage cost is determined by the length (in bits) of the cryptographic key that must be securely stored for each algorithm. By analyzing these costs across different algorithms, this study aims to provide insights into their performance and resource requirements, facilitating informed decision-making in cryptographic protocol selection and implementation.

1 Libraries

1. **pwn** This library is a collection of utilities for binary exploitation, primarily focused on writing exploits for security challenges. The xor function performs XOR operations on byte strings.
2. **pyaes** This library provides AES encryption and decryption functionality. It allows you to perform AES encryption using various modes (such as ECB, CBC, CTR) and with different key sizes.
3. **os, random, secrets** These are Python standard libraries used for generating random data, managing operating system-dependent functionality, and working with secrets (secure random numbers).
4. **cryptography :** This library is used for various cryptographic operations, including RSA encryption and decryption, SHA-256 hashing, HMAC generation, ECDSA (Elliptic Curve Digital Signature Algorithm), and various padding schemes. It provides high-level APIs for working with cryptographic primitives.
5. **hashlib :** This standard Python library provides hashing algorithms like SHA-256. It allows you to compute hash values of data.
6. **hmac :** This library provides functions for creating message authentication codes using cryptographic hash functions, such as HMAC-SHA256.
7. **CMAC and AES from Crypto.Hash and Crypto.Cipher :** These are modules from the pycryptodome library, which provides cryptographic functionality. CMAC is used for AES-CMAC (Cipher-based Message Authentication Code), and AES is used for AES encryption and decryption.

2 Cost Calculation for each of the algorithms :

Computational cost is the time that it takes for the algorithmic half part to get completed for eg in the authentication encryption part , time that is used to generate the cipher text and the tag is called a atomic process time of which we calculate . the communication cost refers to the length (in bits) of the packet communicated from the sender to the receiver in the algorithm, and the storage cost refers to the length (in bits) of the key that needs to be securely stored in this algorithm.

2.1 Symmetric-Key encryption and decryption

- **Communication Cost** Length of the Cipher Text alone .Symmetric key crryptography with the ctr mode or with the cbc mode generates the same length of the ciphertext as that of the plaintext .
- **Storage Cost** Storage cost is the cost of the key that is being stored in the machine . It is 128 bit for this specific case .

2.2 Asymmetric-Key encryption and decryption

RSA , ECC algos

- **Communication Cost** RSA-2048 encryption, the RSA modulus is 2048 bits long. When plaintext data is encrypted using RSA with a 2048-bit key, the resulting ciphertext will also be 2048 bits long. This means that for every block of plaintext data encrypted using RSA-2048, the corresponding ciphertext will require 2048 bits to be transmitted over the communication channel.
- **Storage Cost** Size of the RSA keys is the size of the public key and private key of the sender . Both of which are of 2048 bits size . Very expensive thus .

2.3 Symmetric-Key Authentication and verification

AES-128-CMAC ,SHA3-256-HMAC

- **Communication Cost** in AES-128-CMAC the authentication tag generated is typically 128 bits long, which corresponds to the key size of the AES cipher used internally. When a message is authenticated using AES-128-CMAC, the authentication tag (128 bits) and the original message are sent together as a packet over the communication channel. In SHA3-256-HMAC, the authentication tag generated is typically 256 bits long, which corresponds to the output size of the SHA3-256 hash function.
- **Storage Cost** AES-128-CMAC : Symmetric key of 128 bit is stored SHA3-256-HMAC : Symmetric key of 128 bit

2.4 ASymmetric-Key Authentication and verification

RSA-2048-SHA3-256-SIG ,ECDSA-256-SHA3-256-SIG

- **Communication Cost** RSA-2048-SHA3-256-SIG for authentication, the communication cost includes the size of the RSA signature(2048 bits) after the message is hashed with the SHA3-256 . and the size of the hash message (256bit) that is sent . ECDSA-256-SHA3-256-SIG for authentication, the communication cost includes the size of the ECDSA signature (512 bits) and the size of the SHA3-256 hash (256 bits) of the message.
- **Storage Cost** Size of the RSA keys is the size of the public key and private key of the receiver . Both of which are of 2048 bits size . Very expensive thus .

2.5 Symmetric key Encryption and authentication

AES-128-GCM

- **Communication Cost** In AES-128-GCM, the authentication tag size is 128 bits, corresponding to the size of the authentication key used. Plus the size of the Cipher text that is the size of the plaintext .
- **Storage Cost** Storage cost is the cost of the key that is being stored in the machine . It is 128 bit for this specific case .

3 Generating the test cases

I generated the testcases using the code snippet given below and also calculated the graph using it

```python
def generate_plain_text(size):
    return ''.join(random.choices(string.ascii_letters + string.digits + ' ', k=size))

test_cases = []
sizes =[]
for i in range (50):
    sizes.append((i+1)*100)

for size in sizes:
    plaintext = generate_plain_text(size)
    test_cases.append(plaintext)
```

Figure 1: Random test cases generation

```python
auth_tag_cmac_list = []
verify_cmac_tag_list = []
for i in range(len(test_cases)):
    start_time = time.time()
    auth_tag_cmac = crypto_instance.generate_auth_tag('AES-128-CMAC-GEN', symmetric_key, tes
    end_time = time.time()
    auth_tag_cmac_list.append((end_time - start_time)*1000)
    start_time = time.time()
    is_verified_cmac = crypto_instance.verify_auth_tag('AES-128-CMAC-VRF', symmetric_key, te
    end_time = time.time()
    verify_cmac_tag_list.append((end_time - start_time)*1000)

plt.figure(figsize=(10, 6))
plt.plot(sizes, auth_tag_cmac_list, label='Auth Tag Generation Time (AES-128-CMAC-GEN)')
plt.plot(sizes, verify_cmac_tag_list, label='Auth Tag Verification Time (AES-128-CMAC-VRF)')
plt.xlabel('Size of Plaintext Message (bytes)')
plt.ylabel('Time (ms)')
plt.title('Time Taken for Authentication Tag Generation and Verification vs. Size of Plainte
plt.legend()
plt.grid(True)
# plt.show()
# # # Verification
# print(f"isverified :{is_verified_cmac}")
```

Figure 2: graph generations

4 Observations and the GRAPHS

4.1 Observations

- **Could not draw the RSA graphs for different lengths of the plaintext , as the library has an upper bound on the number of the bits it supports for the general case RSA algorithm . THis bound was about 200*8=1600 bits**
- **There are sudden Spikes in the graphs , could be caused by external factors such as interrupts, context switches, or fluctuations in system load .**
- **There are sudden Spikes in the graphs , could be caused by external factors such as interrupts, context switches, or fluctuations in system load .**
- **here are sudden Spikes in the graphs , could be caused by external factors such as interrupts, context switches, or fluctuations in system load .**
- **AES-CBC-128 encryption time does not increase on increasing the nummber of blocks , while the decryption time increases significantly**
- **AES-CTR-128 is the most expensive algorithm out of the all the algos in terms of the time (RSA I could not calculate because of the library issue)**
- **The time taken for the tag bit generation and that for the verification is almost the same for the SYMMETRIC key , because we are just repeating the tag generation and comparing the tag**
- **HASHINIG bases algo , that hash the message bit and then apply the ECDSA and RSA .The time associated with them tend to remain same , but very slowly**

Table			
Algorithm name	Time(ms)	Packet_length (bits)	Key_lengths (bits)
AES-128-CBC-ENC	0.10251	128*(No. of blocks)	128,128
AES-128-CBC-DEC	0.62847		
AES-128-CTR-ENC	1.04451	128*(No. of blocks)	128,128
AES-128-CTR-DEC	0.95176		
RSA-2048-ENC	1.166820	2048	2048,2048
RSA-2048-DEC	1.981258		
AES-128-CMAC-GEN	0.7190704345703125	128*(No. of Blocks) + 128	128,128
AES-128-CMAC-VRF	0.4990100860595703		
SHA3-256-HMAC-GEN	1.310586929321289	256 bit tag + Messagebits	128,128
SHA3-256-HMAC-VRF	0.766754150390625		
RSA-2048-SHA3-256-SIG-GEN	2.25377	2048 + 256	2048,2048
RSA-2048-SHA3-256-SIG-VRF	1.29032		
ECDSA-256-SHA3-256-SIG-GEN	0.80132	512 + 256	256,256
ECDSA-256-SHA3-256-SIG-VRF	0.981330		
AES-128-GCM-GEN	0.945329	128*(No of blocks) + 128+	128,128
AES-128-GCM-VRF	0.89168		

increase . This is because the operation is only on the 256 bit while the hashing of the message bits is pretty fast does not create much difference.

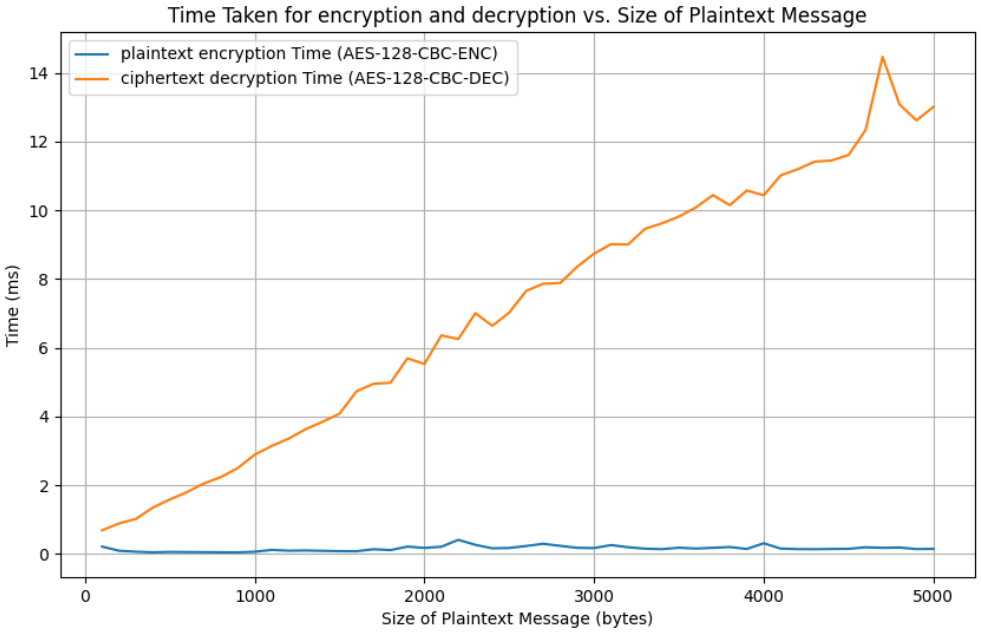

Figure 3: The encryption scheme that is being followed.

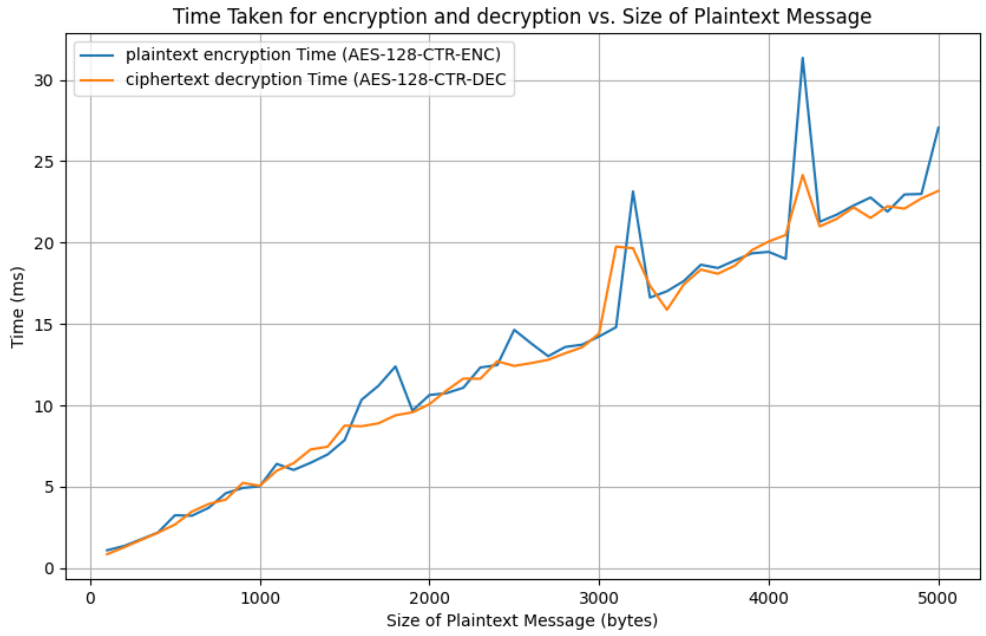

Figure 4: The encryption scheme that is being followed.

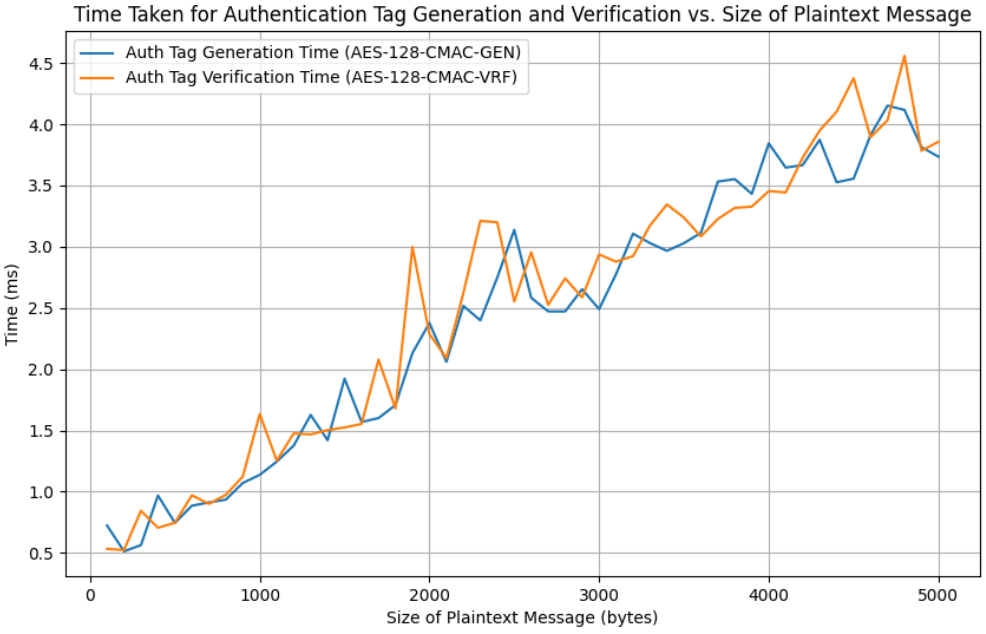

Figure 5: The encryption scheme that is being followed.

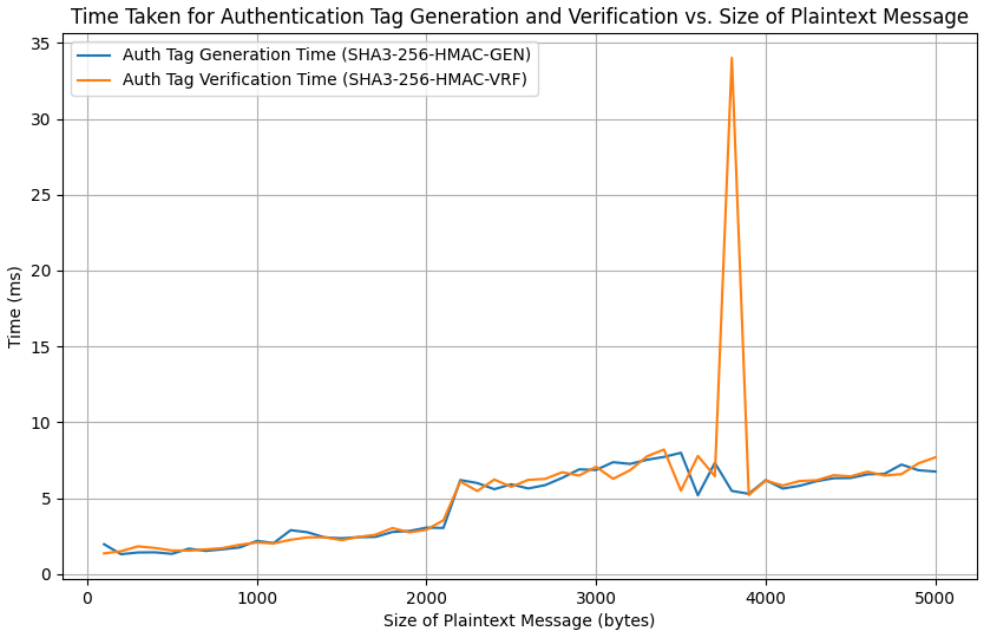

Figure 6: The encryption scheme that is being followed.

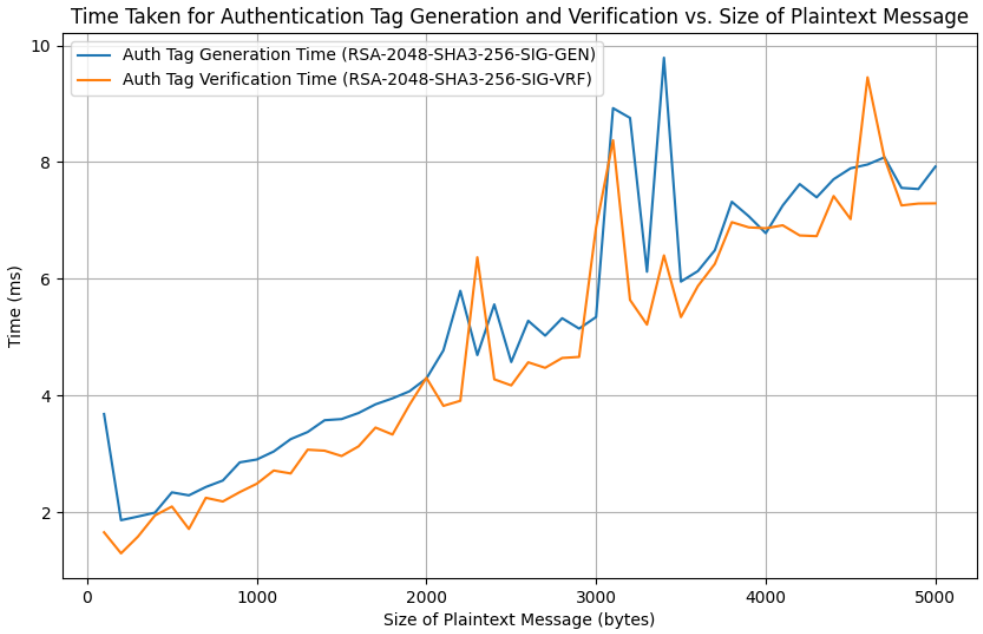

Figure 7: The encryption scheme that is being followed.

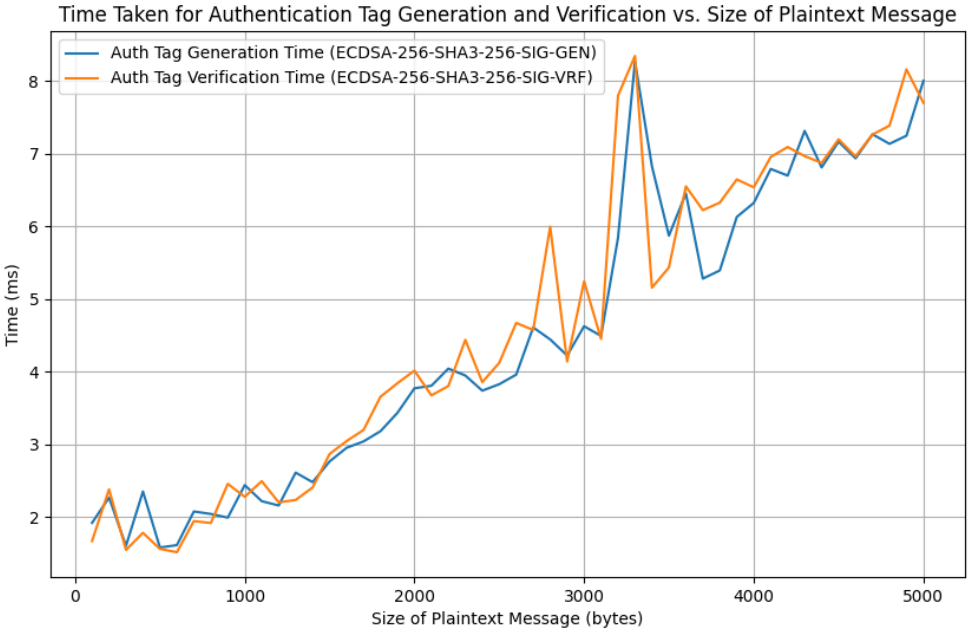

Figure 8: The encryption scheme that is being followed.

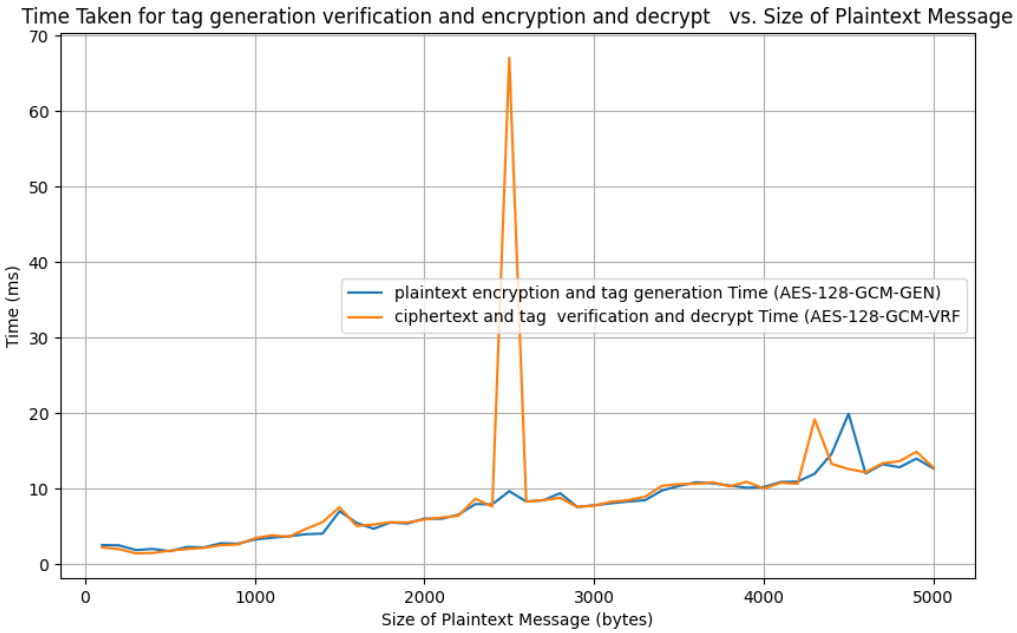

Figure 9: The encryption scheme that is being followed.