NSS Assignment - 2

Name: Suraj Kiran Mate
Entry No : 2021JCS2387

A] Code Explanation:

In this assignment we are comparing different encryption schemes considering different parameters. In this we have used **AES-CBC, AES-CTR, RSA** as the encryption and decryption schemes. Also we have used different authentication methods **AES-CMAC, AES-HMAC, RSA-SHA3** and **ECDSA**. All these schemes are compared in terms of different parameters like key size required, time taken for execution and packet size generated after the operation.
In this complete discussion we will go step-by-step first starting with actual discussion of the code. The links for all the libraries used are given at the end of this report.

Starting with the code we will first discuss all the libraries that we have installed in this code. So the first library we have installed is **cryptography** also we have installed **pycryptodome**, we are importing certain modules from both of these libraries. We are also importing **OS** for random number generation. We have also installed **ECDSA** which is **E**lliptic **C**urve **D**igital **S**ignature **A**lgorithm for signing the message using elliptic curve key generation and signing algorithm.

Starting with a code in the first function **generate_keys** we have to generate keys for different encryption schemes. In symmetric key encryption we know that for both encryption as well as decryption we require only one key so we are generating this random **symmetric_key** of 128 bit size using **os.urandom** function. For RSA we actually require two keys public key and private key, so to generate both these keys we are using **RSA** module which we have imported from pycryptodome library.
RSA.generate(key_size) will generate the private key and to generate the corresponding public key we just need to write **private_key.public_key()**. We will generate both the private_key and public_key pairs for both sender as well as receiver. We are using **ECDSA** library to generate both public and private keys for elliptic curve cryptography. During generation of key for elliptic curve we have to mention the curve for key generation.

In the second function **generate_nonces** we are generating nonces by using OS module. For this we write **os.urandom(size_in_bytes)**. We could also use **os.random()** function in os module but this function does not generate pure random numbers it generate pseudo random numbers which does not satisfy the requirements for cryptographic use. Because if the random number generation gets compromised then the complete cryptographic scheme will get compromised. Nonces are one time generated random numbers which we don't need to keep secret after their use.

In the third function **encrypt** we have to encrypt the plaintext using different encryption schemes like **AES-CBC**, **AES-CTR** and **RSA**. For all these encryptions we are using the modules from pycryptodome and cryptography libraries. Before encryption we have to ensure that the plaintext we are going to pass to these functions should be in byte form because in our case the plaintext which is passed is given in string format so we have to first encode it into byte form and then encrypt it. Thus the ciphertext which will generate will also be in byte form.

In the fourth function **decrypt** we have to decrypt all the cipher text that we have generated in the previous function. In both AES CBC and CTR modes we are using the same symmetric key for both encryption as well as decryption but in RSA mode we are using public key of receiver for encrypting the data but the private key of receiver for decrypting the data. Also for all decryptions we are using the both cryptography and pycryptodome libraries.

In the fifth function **generate_auth_tag** we are generating authentication tags on the plain texts that we have passed. Here also we have to make sure that the plaintext that we are passing is in byte form and not in string form. We are generating this authentication tax using four different functions: AES-CMAC, SHA3–HMAC, RSA-SHA3, ECDSA-SHA3. SHA3 is getting used just to generate the message digest. For authentication tags also we are using pycryptodome library.

In the sixth function **verify_auth_tag** we are verifying all the authentication tax that we have generated in the previous function. As we already told that we are using pycryptodome library for both authentication tax generation as well as verification. The speciality of this library is, if the authentication tag is valid then it returns true but if the authentication tag is not valid or it is not matching with what we have generated before then it will raise exception. To take extra care of these exceptions we are using try and except functionality in python so if the exception is not raised then we are returning **True** else we are returning **False**.

In the next function **encrypt_generate_auth** we are going to encrypt as well as authenticate the plaintext using **AES–GCM**. Here also we are using cryptography library. In this we have to pass both encryption as well as authentication-generation keys. The encryption key will encrypt the plaintext and authentication generation key will authenticate the plaintext thus ciphertext and authentication tag will be the output of this function.

In the last function **decrypt_verify_auth** we have to both decrypt as well as authenticate the given ciphertext. Generally we first authenticate the ciphertext and if the authentication is found to be correct then only we proceed to decrypt it otherwise we do not decrypt it. While checking the authentication it also raises the exception if the authentication tag is invalid so we also have to take care of it using try and except. After decryption the generated plain text is in byte form and we have to return it into string form. We have to note here that if the authentication failed then we are going to return the empty string in byte form, otherwise we will return the normal plaintext.

B] Result Table:

Algorithm	Key Length (bits)	Execution Time (mili sec.)	Packet Length (bits)
AES-128-CBC-ENC	128	42.29	640 (5120 bits)
AES-128-CBC-DEC	128	0.11	
AES-128-CTR-ENC	128	39.31	640 (5120 bits)
AES-128-CTR-DEC	128	0.214	

Algorithm	Key Length (bits)	Execution Time (mili sec.)	Packet Length (bits)
RSA-2048-ENC	2048	1.367	256 (2048 bits) (for 16 bytes message)
RSA-2048-DEC	2048	3.832	
AES-128-CMAC-GEN	128	43.8	633 + 16 = 649 (5192 bits)
AES-128-CMAC-VRF	128	0.0441	
SHA3-256-HMAC-GEN	256	0.840	633 + 64 = 697 (5576 bits)
SHA3-256-HMAC-VRF	256	0.1602	
RSA-2048-SHA3-256-SIG-GEN	2048	4.247	633 + 256 = 889 (7112 bits)
RSA-2048-SHA3-256-SIG-VRF	2048	0.7262	
ECDSA-256-SHA3-256-SIG-GEN	256	2.177	633 + 96 = 729 (5832 bits)
ECDSA-256-SHA3-256-SIG-VRF	256	6.33	
AES-128-GCM-GEN	128	1.066	649 (5192 bits)
AES-128-GCM-GEN	128	0.3428	

Table 1: Results of all algorithms

C] Pros and Cons Discussion:

From all the result that we have generated and encapsulated into the table we are sure that there is slight variation in the parameters of these algorithms but some of the algorithms are varying with larger margin.

The bar graph that we have drawn below gives the clear idea about the time required for both encryption as well as decryption for different algorithms. We plotted the time on y-axis and placed different algorithms on x-axis.

We know that if the process takes too much time for computation means it takes more CPU cycles and we can directly relate it as computationally heavy. From the chart below we can see that AES-CBC and AES-CTR both the modes take comparatively more time for encryption as well as decryption than RSA-2048.

Chart 1: Time Comparison for different Algorithms

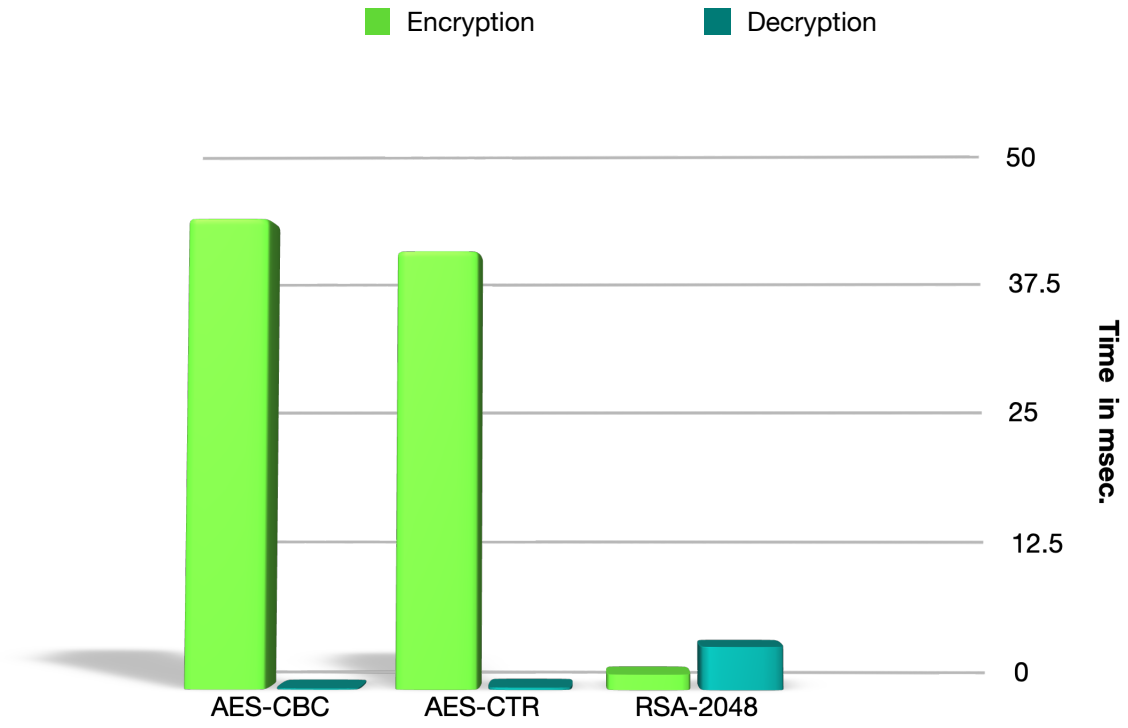
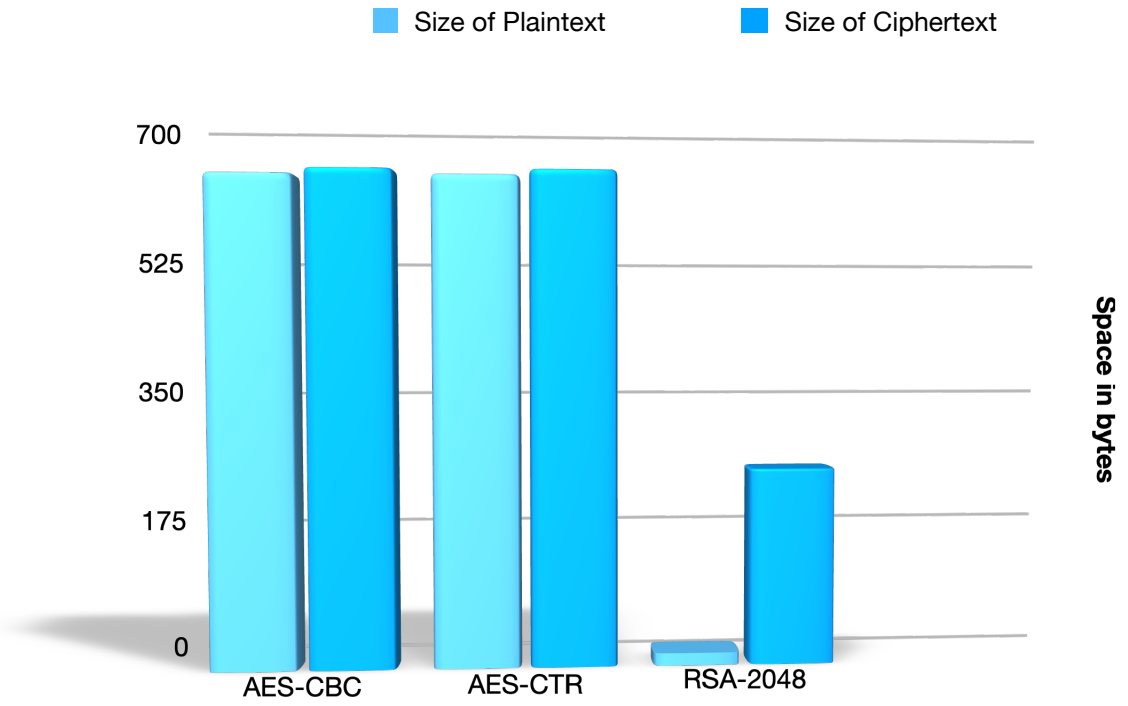


Chart 2: Space Comparison for Different Algorithms



But this is not a clear picture due to the abstraction we made here, one thing to note here that the plaintext that we fed to AES-CBC and AES-CTR is 633 bytes in length. While the plaintext that we fed to RSA was only 16 bytes in length. Due to such small plaintext we can see that the time taken by RSA for encryption is comparatively less than both AES methods.

Also this library is structured in such way that we need to pass the plaintext in bytes form for both AES methods while we need to pass plaintext in string form for RSA method. Thus extra computation is done for converting the plaintext in byte form which also contribute to time consumed.

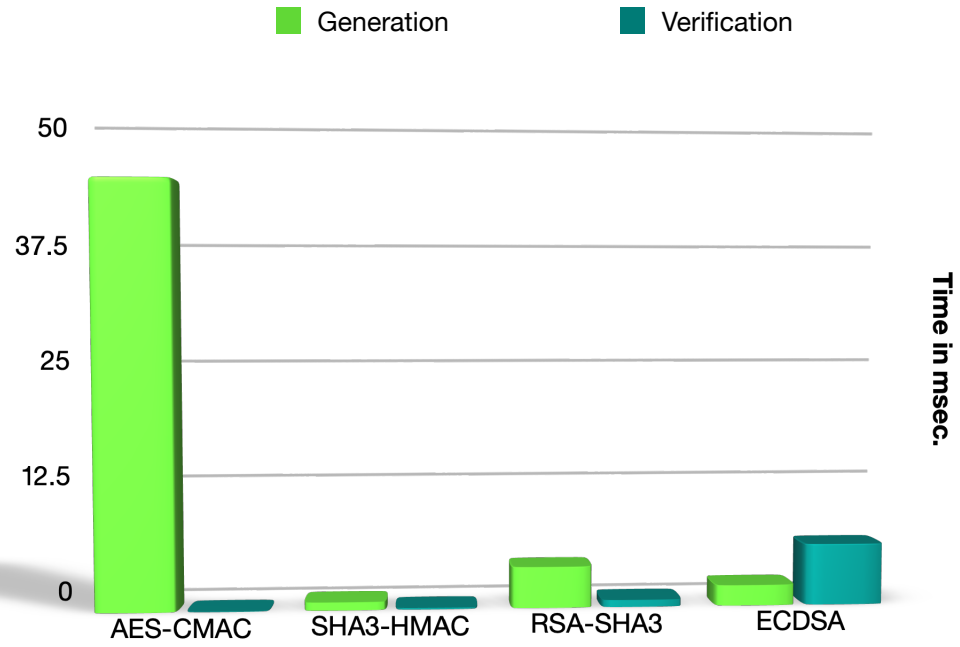
Generally the RSA encryption methods are comparatively light compared to the decryption methods due to the size of the public exponent which is less than the size of private exponent. The private exponent used in RSA is bigger than the symmetric key for AES so that the time taken just to decrypt this 16 byte plaintext is more than the time taken by both the AES methods.

Along with this the cyphertext generated by both AES methods was only 640 bytes when we compare it with the size of plaintext given in input ie. 633 bytes but the ciphertext generated by RSA on the 16 byte plaintext was 256 bytes long. This also tells us that RSA algorithms are memory heavy as well. Before this we only knew theoretically that the symmetric algorithms are good when there are CPU as well as memory constraints but now we proved it practically.

So symmetric schemes are computationally light but there is always an overhead of sharing the symmetric key from sender to receiver and if the key gets compromised then the complete communication will get compromised but in asymmetric schemes like RSA we don't have to care about sharing the keys. Thus in the context of security asymmetric schemes takes an edge over symmetric schemes.

Now we are going to compare different digital signature algorithms for message authentication at receiver end. We will again compare the same way, we will first compare the time required by each algorithm and then the space requirement.

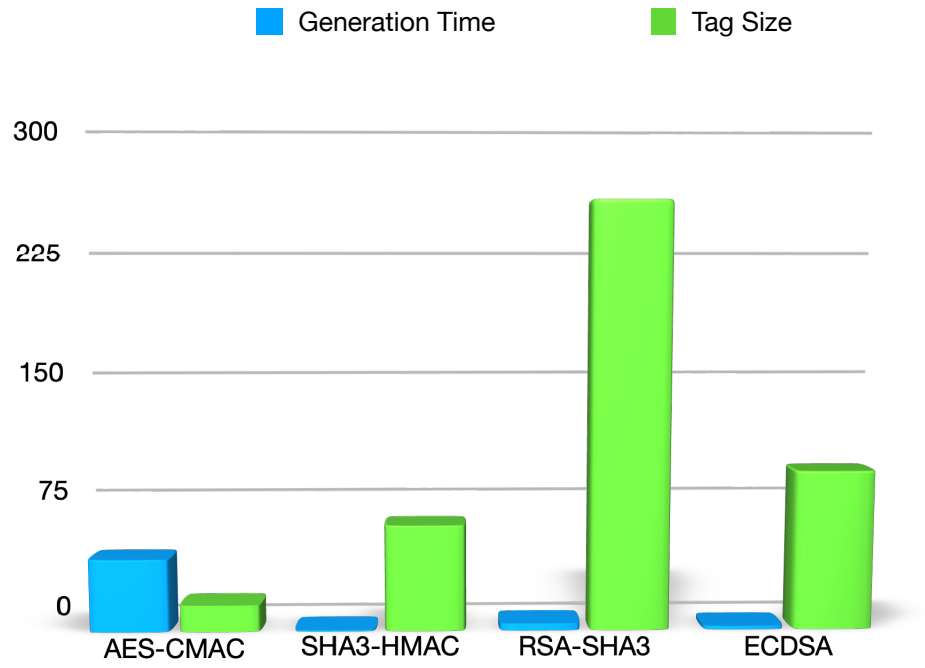
Chart 3: Time comparison of authentication Algorithms



But during signature generation as well as verification the picture is different. We can see that the time taken by AES for generating authentication tag is almost 10 times the time taken by RSA for the same. Hence this was unexpected by some symmetric method. Also we have used the same message length for authentication tag generation and verification still time then by AES for generating the auth_tag is more than RSA and ECDSA. But during verification of authentication tag AES is faster than RSA and ECDSA. ECDSA is slowest in this terms. We ran these functions multiple times and average out the time observations.

But moving forward to compare the length of authentication tags, we can see that the size of tag generated by AES is only 16 bytes, tag size of SHA3-HMAC is 64 bytes, tag size of RSA-SHA3 is 256 bytes and tag size of ECDSA-SHA3 is 96 bytes.

Chart 4: Time and Space comparison of Authentication Algorithms



Even though the RSA authentication is not heavy in terms of CPU cycle consumption but it is surely heavy in terms of memory consumption. ie. space required for RSA auth_tag is almost 16 time the space requires for AES generated auth_tag. The tag size of ECDSA is also more than the tag size of AES-CMAC and SHA3-HMAC. Thus this also tells that asymmetric schemes like RSA and ECDSA takes more space compared to symmetric schemes like AES-CMAC and HMAC.

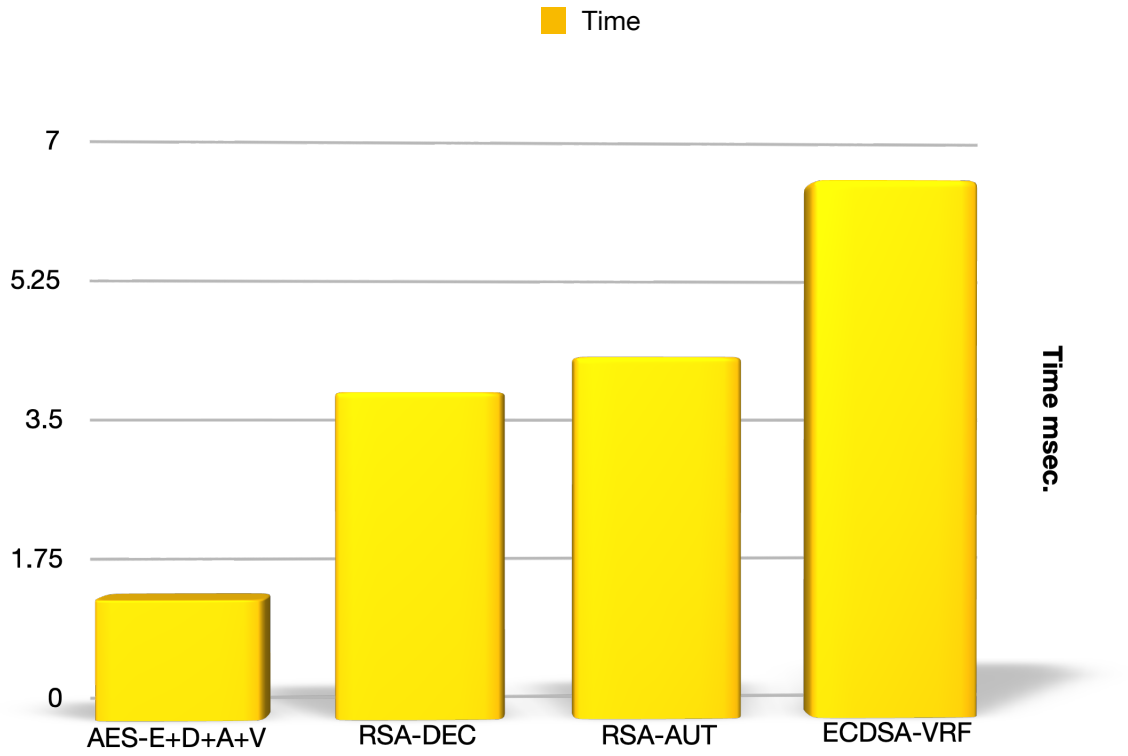
The space taken in above case for RSA is more than the space taken by combined AES-CMAC and SHA3-HMAC. But still the same advantage of avoiding the overhead of sharing the key persists here as well. Because in symmetric schemes we require the same key for both generation as well as authentication but in asymmetric schemes we require private key for generation of authentication tag and public key for its corresponding verification.

The last method is combined encryption and authentication which we normally require when dealing with real end-to-end communication. Here we are using AES-128-GCM which is a symmetric key method. Due to both simultaneous encryption and authentication it is obvious that the time taken by the algorithm is more. But surprisingly it takes only **1.066 msec** for both encryption as well as generation of authentication tag on the message of length 633 bytes and takes only **0.3428 msec** for combined decryption and verification of authentication tag.

Comparing this last method with some independent asymmetric key encryption and authentication methods can give us the better picture of how good the symmetric algorithms are when it is about CPU and memory space.

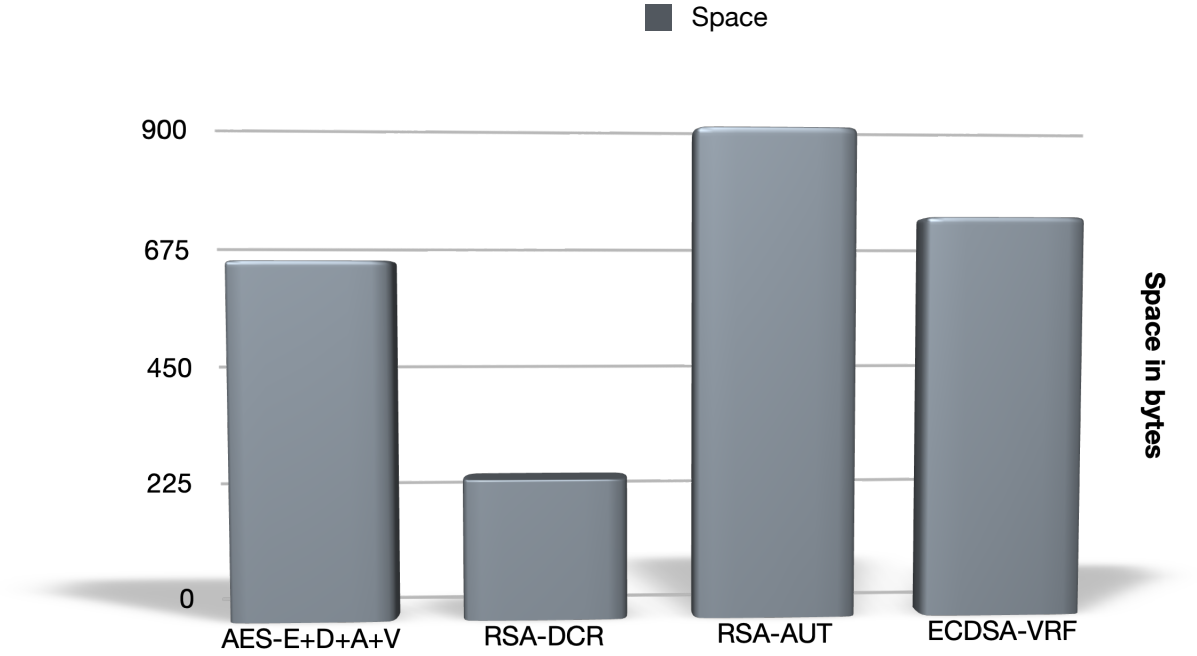
We have drawn the combined time and space comparison for different methods comparing with the time taken for AES encryption and authentication.

Chart 5: Time comparison of different methods with AES combined encryption and authentication



Here **AES-E+D+A+V** represents that the scheme is AES along with encryption, decryption, authentication and verification combined. Hence complete communication terminology is packed into one. We can observe that the time taken by this combined method is almost equivalent to independent RSA encryption and independent RSA authentication but it is very less compared to independent elliptic curve verification. Thus the AES is this much efficient.

Chart 6: Space comparison of different methods with AES combined encryption and authentication



In terms of space requirement the space taken is almost equal to RSA authentication and ECC verification. We also have to note here that the bars we are seeing for RSA decryption is small but they are not considering the decryption of complete 640 byte ciphertext but just 256 bytes ciphertext. If we could draw it on the same data then it would be a different story for RSA as well. Hence the clear win for AES in combined encryption, decryption, auth_generation and auth_verification. Thus the computation and communication costs required for asymmetric schemes are always more than symmetric methods.

Hence this proves that if the end to end communicating party is limited with computational resources then they should go for symmetric key encryption and authentication while if the computational resources or not a problem then the party should surely go for asymmetric encryption and authentication. The advantage with asymmetric key algorithms is that there is no compromise with security under the name of key sharing. Also no need to change the key for every session of communication.

Thank You

D] References:

1. **https://cryptography.io/en/latest/** - Documentation of cryptography library
2. **https://pypi.org/project/ecdsa/** - Documentation of ecdsa
3. **https://www.pycryptodome.org/en/latest/** - pycryptodome documentation