README

By: Anmol Agarwal (2018CS10327)

Libraries used:

- cryptography (version 36.0.1)
- os
- time

Python version: Tested with python 3.6, 3.8

Reference: Welcome to pyca/cryptography — Cryptography 37.0.0.dev1 documentation

Files:

1. execute_crypto.py
2. example_test.py
3. original_plaintext.txt
4. readme.txt
5. readme.pdf
6. setup_env.sh
7. measurements.xlsx

About Code:

1. execute_crypto.py
   Contains the implementation of all functionalities from key/non generation to (authenticated) encryption/decryption and authentication.
   1. generate_keys():
      - symmetric key generated using os.urandom()
      - private_key generated using generate_private_key(), public_key derived from private_key using public_key() function.
      - Private/public key objects were serialized and de-serialized for transmission from sender to receiver.

   2. generate_nonce():

      - Initialization vector used for aes-cbc
      - Nonce generated only for aes-ctr, aes-gcm. Rest didn't require use of nonce.

   3. encrypt(): steps of any algorithm are given as follows

- Plaintext is converted to bytes.
- For AES-CBC, PKCS7 padding is used to pad plaintext to 128 bits. For RSA, even though 128 bit symmetric key used, I have given generic implementation where plaintext of any length is broken down in 128 bit chunks and then ciphertext is given as output.
- For asymmetric protocols, encryption is done using public key of receiver.

4. decrypt():

- Ciphertext is already given in bytes.
- For AEC-CBC, padding is removed on decrypted ciphertext to get plaintext. For RSA ciphertext is divided in size of 2048 bits, and then decrypted, and finally plaintext is constructed back.
- For asymmetric protocols, decryption is done using private key of receiver.

5. generate_auth_tag():

- Plaintext (converted to bytes) is given as input and signature is generated.
- In symmetric protocols, symmetric key is used for generating signature.
- In asymmetric protocols, private key of sender is used for generating signatures.

6. verify_auth_tag():

- In symmetric protocols, symmetric key and plaintext is used for verification.
- In asymmetric protocols, plaintext and public key of sender is used for verifying signatures.

7. encrypt_generate_auth():

- Here nonce and symmetric key is used for aes-gcm. The output of the encrypt() function is ciphertext along with 16 byte auth_tag appended at the end (as given in documentation).

8. decrypt_verify_auth():

- Here we take ciphertext, auth_tag and symmetric key as input, ciphertext is concatenated with auth_tag at the end and then

given to decrypt function to generate plaintext and verify in parallel.

For authentication, InvalidSignatureException is caught if receiver does not verify the input.

2. run_algorithms.py
   Separately calls the functions from execute_crypto.py to carry out measurements of computation, communication, and storage costs. Contains three functions:
   1. run_encrytion: runs encryption/decryption of algorithm given as input and returns measurements.
   2. run_authentication: runs authentication algorithms given as input and returns measurements.
   3. run_authenticated_encryption: runs aes-gcm algorithm and returns measurements.

Table containing Cost Measurements

Algorithm	Key Length (bits)	Execution Time (ms)	Packet Length (bits)
AES-128-CBC-ENC	128	0.0815	
AES-128-CBC-DEC	128	0.0397	5120
AES-128-CTR-ENC	128	0.0753	
AES-128-CTR-DEC	128	0.0334	5072
RSA-2048-ENC	3608	0.1034	
RSA-2048-DEC	13432	3.3706	2048
AES-128-CMAC-GEN	128	0.0618	
AES-128-CMAC-VRF	128	0.0308	5200
SHA3-256-HMAC-GEN	128	0.0465	
SHA3-256-HMAC-VRF	128	0.0222	5328
RSA-2048-SHA3-256-SIG-GEN	13432	3.4617	
RSA-2048-SHA3-256-SIG-VRF	3608	0.0993	7120
ECDSA-256-SHA3-256-SIG-GEN	1816	0.1362	
ECDSA-256-SHA3-256-SIG-VRF	1424	0.1130	5648
AES-128-GCM-GEN	128	0.0374	
AES-128-GCM-VRF	128	0.0224	5200

Note:

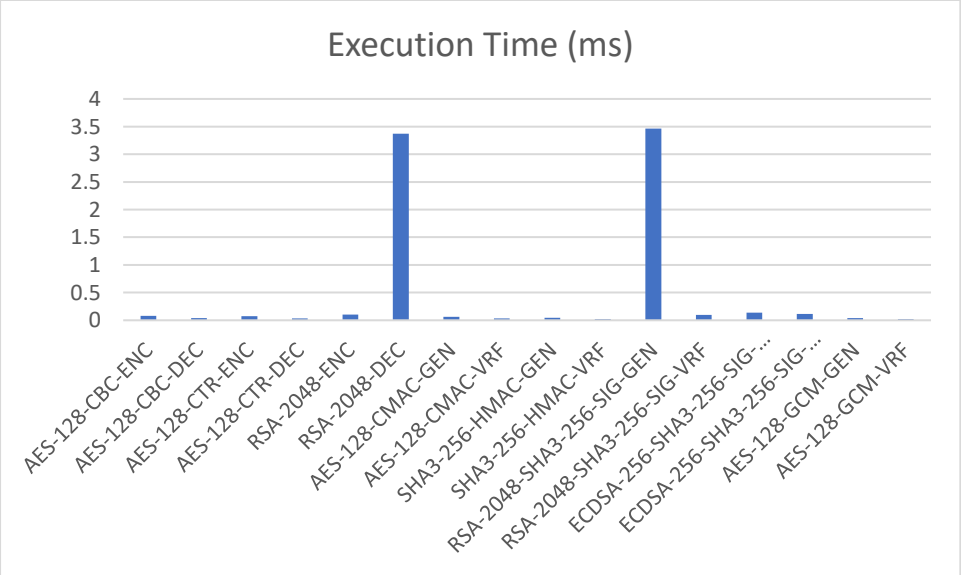
1. for ECDSA, packet length or signature length were varying slightly as seen from the table. Similarly, RSA public key (serialized) length was also varying.

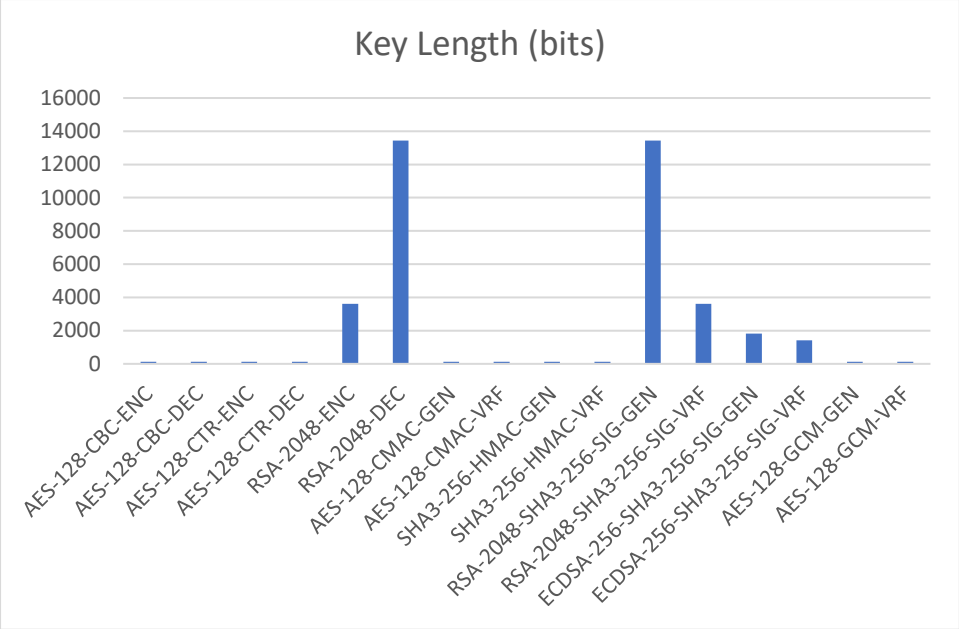
2. For RSA-2048, I also tried with large plaintext as used in other algorithms and the findings were:

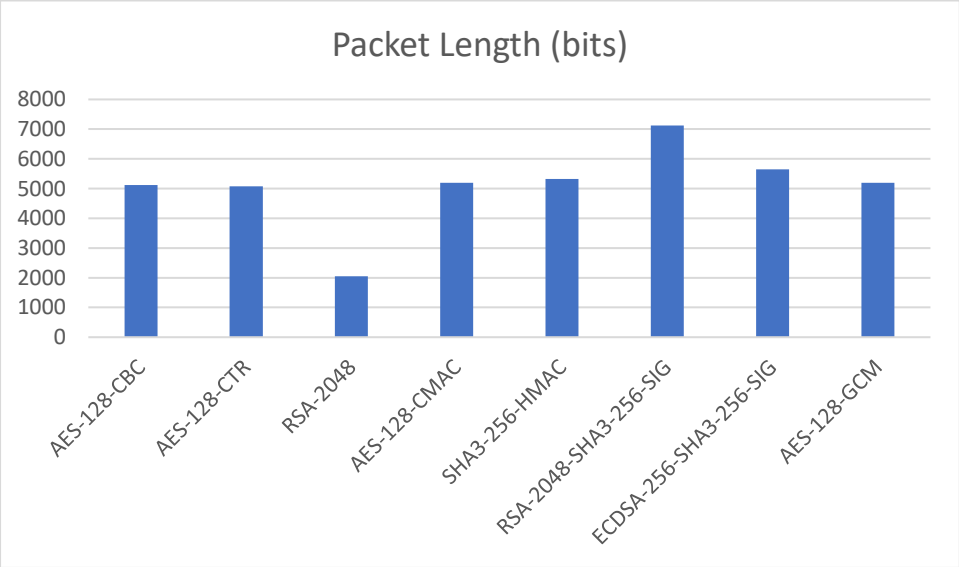
Algorithm	Key Length (bits)	Execution Time (ms)	Packet Length (bits)
RSA-2048-ENC	3608	0.2527	
RSA-2048-DEC	13400	5.4106	1280

Plots

(Next Page)







Overall Findings

1. Execution Time: execution time of RSA during decryption and signature generation was very high.
2. Key Length: Since I was serializing private and public keys, instead of using private / public key objects (as happens in practical scenarios) the key length of asymmetric protocols was high.
3. Packet Length: RSA signature had the highest packet length in authentication but the lowest in encryption. Other algorithms had similar packet lengths.

Findings: Comparing Algorithms

Encryption

1. **AES-CBC:** Execution time was low since it was a symmetric key protocol, but packet length was high.
2. **AES-CTR:** Similar findings as in AES-CBC.
3. **RSA-2048:** It had high execution time for both encryption and decryption and this is expected since its an asymmetric key protocol. But packet length was low compared to above two algorithms.

Authentication

1. **AES-CMAC:** Had low execution time because of the symmetric nature. Moreover, it had the lowest packet length, so lowest storage overhead.
2. **SHA3-256-HMAC:** Had low execution time, had lowest packet length.
3. **RSA-2048-SHA3-256-SIG:** Had highest overhead among all algorithms in execution time, key length (because serialized key was shared, but still the key size is 2048 bits) and packet length.
4. **ECDSA-256-SHA3-256-SIG:** Execution time was higher symmetric protocols, but much lower than RSA. And packet length was comparable with AES-CMAC. But key length was higher (because of serialization).

Authenticated Encryption

1. **AES-GCM:** It had low encryption and authentication time, low packet length and key length. But now we need to share nonce along with signature and ciphertext.

Advantages and Disadvantages of Each Algorithm

1. **AES-CBC:** It has low costs, even though encryption is not parallelizable because of the sequential nature of the chain, decryption is parallelization by simply getting the previous ciphertext, current ciphertext and the key to get current plaintext. Moreover, IV needs to be shared across the channel.
2. **AES-CTR:** but it has an additional advantage that nonce need not be hidden and it can be parallelizable, because once we know nonce value, we can get counter values to use for different blocks the ciphertext.

3. **RSA-2048:** It is asymmetric key protocol and has high overhead. And message needs to be broken down to fit 2048-bit key size, in other words, it needs to be less than n (the product of two prime numbers).
4. **AES-CMAC:** It is hard to compromise confidentiality by changing one block of plaintext because it gets propagated in the chain. But it can be vulnerable when attacker adds few bits after message, so that the receiver gets the wrong hash value.
5. **SHA3-256-HMAC:** This is faster than CBC mode for CMAC because we don't need to using chaining to get the digest. Even though SH3-256 is slightly slower, there are many hash functions which we can use.
6. **RSA-2048-SHA3-256-SIG:** It has extremely high overhead in execution time compared to other authentication algorithms.
7. **ECDSA-256-SHA3-256-SIG:** As per NIST, ECDSA requires lower key length compared to RSA for equivalent security. Moreover, it also has very low computation cost compared to RSA.
8. **AES-GCM:** It is parallelizable and hence efficient. Moreover, it is considered more secure than AES-CBC, because AES-CBC and AES-CMAC are vulnerable when attacker adds bits/padding to end of the message unlike AES-GCM.