[CNT5410] Report

Xiaohan Wang

February 2, 2016

1 Introduction

This is the report for CNT5410 Computer and Network Security course. I conducted the required encryption/decryption or hash operations including AES 128, AES 256, RSA 1024, RSA 4096, MD5, SHA1 and SHA256. My programming and experimental environment is based on Windows 8.1. I use Virtual Box to manage a virtual machine of Ubuntu 14.04 . I use Eclipse IDE to program and tune the code and generate a makefile for my program. All of my work is done by myself but I read the libgcrypt library manual website thoroughly and adopted some standard sample codes posted on webpage such as initializing cryptographic environment and freeing memory that is allocated. I also searched related materials online extensively because some of the details of the libgcrypt are not explicitly elaborated on the manual site and there are almost no example codes as references.
I tested AES128 and AES256 with 100MB input file first, however the time consumption was too large. For a 100MB mp4 file, a single AES256 decryption and encryption operation will spend 13-14 hours to finish. 100 runs will need estimatedly 50 days. So if I still used 100MB file as an input, it is impossible to finish the experiment within 14 days (from assignment posted day to due date). The reason for this may be that my VM environment cannot support such highly time-consumed operation. Further more, two RSA operations only support an input file that contains characters no more than 128 bytes. I have tested and dived into library source code to verify that. The conclusion is that to measure 100MB data encryption/decryption is infeasible for me following homework instruction. In order to obtain a less acceptable yet better than zero result, I created a file randomly less than 128-byte (for RSA input constraint) as input.

2 Challenges and Pitfalls

The followings are major challenges I met during implementing the program.

2.0.1 AES Operation Handler

For AES enciphering and deciphering algorithm, handler data structures (gcry_cipher_hd_t) are needed for the called function. This is used to store necessary information for decryption and encryption. The basic XOR operation for AES algorithm is implemented in an assembly language manner by libgcrypt library (I stepped into the source code and checked it in IDE debugger) and decryption and encryption function need separated handlers. That means in order to encrypt/decrypt plain/ciphered text correctly, two handlers are required, and both of them should be set in the same manner to map an encrypted text to a corresponding plain text. Unfortunately, the manual website doesn't point that out explicitly so it becomes an major obstacle for me at the beginning.

2.0.2 RSA Algorithm

The RSA algorithm is based on number theory, especially the property of the prime number. This feature indicates that the data which RSA works on should be presented in a particular way. RSA function in libgcrypt can't accept an arbitrary stream of data, it needs to convert specific data into so called MPI (Multi-precision Integer) to make the algorithm work (I checked library source code, the author comments that right now his library only accepts "c-string", which is common string consisting of ASCII characters). Further more, the specific data (c-string) which will be converted later is presented in four manners based on the manual. I used "GCRYMPI_FMT_HEX" format, which stores as a string with each byte encoded as 2 hex digits. "GCRYMPI_FMT_USG" is also recommended but converting each byte into unsigned integer will make the string have no fixed size (for example of "12", you don't know if it is original binary data of "0000000100000010" ('1'+'2') or "00001100" ('12')), so former scheme is better. To convert binary data into 2 hexadecimal digits per byte, I preprocessed the read-in file by checking their character ASCII codes and reassigned each 4 bits into '0'-'9', 'a' to 'f' or 'A' to 'F'. This format transformation also was not mentioned on manual web page, I went through extensive additional materials and recapped contents discussed on our lecture to figure out the implementation of the RSA in libgcrypt.

3 Implementation Phase One

In phase one my program calls each required function to encrypt, decrypt or hash input file and checks the correctness of the operation. The snapshots retrieved from Eclipse console are presented below.

1).The image below shows that for AES128 and AES256. We can see the deciphered text is the same as the plain text.

```
AES128 mode:

original text:
313734383633383439323734363537666164656665616263353030343732756 46a6468776879737779766276637662616263 6a616e636e636b6b736e636a7368737661636 16a6263626168616263 0a

ciphered text:
 = 57de7fecd9b9140fb17ecd6dfc011c48398699a89f1f5202034f2d6393d5303e69ecb2b908cfd0e58a5ea0cf12c49c1995f6c0ca272837409c96fa91f7e1b288537470422f19f6b942e8decdb2f0a1

decryption text:
313734383633383439323734363537666164656665616263353030343732756 46a6468776879737779766276637662616263 6a616e636e636b6b736e636a7368737661636 16a6263626168616263 0a

AES256 mode:

original text:
313734383633383439323734363537666164656665616263353030343732756 46a6468776879737779766276637662616263 6a616e636e636b6b736e636a7368737661636 16a6263626168616263 0a

ciphered text:
 = 2ed9eed45c580bca61bcb5e7b12fb2c90aa19903b73bd17ad780919cde5b01d86a0019d298a647c7ba423a0960521b5597bd86b23cc38736fce9ad1769717d0e0a5bd30b9f0c6ad5bf16caef881791

decryption text:
313734383633383439323734363537666164656665616263353030343732756 46a6468776879737779766276637662616263 6a616e636e636b6b736e636a7368737661636 16a6263626168616263 0a
```

2).The image below shows result of RSA 1024, we can see the deciphered text is the same as the plain text. I also printed out the s-expressions of the public and private key.

```
plaintext:
313734383633383439323734363537666164656665616263353030343732756 46A6468776879737779766276637662616263 6A616E636E636B6B736E636A7368737661636 16A6263626168616263 0A

RSA1024:
Public Key:
(public-key
 (rsa
  (n #00CAA24225A8BD49D624AB83417D8D2B4651E71896D34C7994C68CA32416D086FFDD25D7DE08E4AA7175B84CAA4DAAE898F2D71F42DB32A4273921CCD6996CDF41CC5AA0E9FBE0A6F2A5F53A6B83DB5451442D8F7CD07DD7FF0AAC9
  (e #010001#)
  )
 )

Private Key:
(private-key
 (rsa
  (n #00CAA24225A8BD49D624AB83417D8D2B4651E71896D34C7994C68CA32416D086FFDD25D7DE08E4AA7175B84CAA4DAAE898F2D71F42DB32A4273921CCD6996CDF41CC5AA0E9FBE0A6F2A5F53A6B83DB5451442D8F7CD07DD7FF0AAC9
  (e #010001#)
  (d #014091721421DF53F07412E293B90C9C69A328F6BDA58A56CE074E76C9A14E4444849859E9EA27D77DA7BEE98A8354DEA519893AEB89DB93D6798BFF838135BBFA8A53CE879075C240C9B804A98B6BE12FCB8410D2D06801F83F554
  (p #00D6446E74C14AC56185DADCC2D8129310C14F1016E71EDFAA8D443E0C965D9CA92C44F8A7F405CBAA8A1054445975F31170C1617FE8B2F8D00B85340131A79557#)
  (q #00F219C531FDE91D2A613A17E319B76B88C6B113183D4B7CD546DB0C9E5D70CBCEE0A8564E018BF391338964DEC2FD9D8FCB5D406FFB72F51DB3F345FBE1F41FBB#)
  (u #00A3ABF77A8D7419BDEE61CD9574242CBABBBB1C8ABB65EEB940C831FDA6AACD2FC790B1646C0E548E54649999A33002B8B76B17519D2A28324050F41BCA33A790#)
  )
 )

ciphertext:
(enc-val
 (rsa
  (a #00B39C7726DE1EB6D2FD96273A64F1FBD472E50C5FF9907F19E76A60FC020B44CB8578C29CD723DDE3AC6EC339E518A3AE9E23E347CCFB7C62709346A1239F4392D65EECC6E75503EF10F2EF8FA9AC0D37A0C46B6657E59B7F431F2
  )
 )


deciphered text:
313734383633383439323734363537666164656665616263353030343732756 46A6468776879737779766276637662616263 6A616E636E636B6B736E636A7368737661636 16A6263626168616263 0A
```

3).The image below shows the signature I generate by using RSA 1024, it can be verified by libgcrypt func-

tion gcry_pk_verify().

```
RSA digital authentication signature :
(sig-val
 (rsa
  (s #2D6ABEFEBBE171A79D18AFDE4BD1A9D96C5DC22CB1EBAF70FA372D6052BC79D46E29A58D2C7D14CE3162B695A25B699844C26CDC332669AC553938D36BB62D8CD330905559E6A61632AAFFE359CFEFA59191E8D681AD8C0119151E4FB
  )
 )

verify signature, if it is valid then indicating that     program is in good shape!

valid signature!
```

4).The image below shows result of RSA 4096, we can see the deciphered text is the same as the plain text. I also printed out the s-expressions of the public and private key.

```
plaintext:
31373438363338343932373436353766616465666561626335303034373275646A646877687973777976627663766262636A616E636E636B6B736E636A736873766163616A62636261686162630A

RSA4096:
Public Key:
(public-key
 (rsa
  (n #00CDE6BCB9B66E95EF7323E833C7E0C9D7F82D23AC1779B738EFC8CF6C8362DB97E10176A63C5DE3173BCB64BA625539E41AF5B37FF2CF974D6BB15CE714C98022127BDC4D208AECA6176165DB96E30407DC1E34D3F27ADD0032327A2
  (e #010001#)
  )
 )

Private Key:
(private-key
 (rsa
  (n #00CDE6BCB9B66E95EF7323E833C7E0C9D7F82D23AC1779B738EFC8CF6C8362DB97E10176A63C5DE3173BCB64BA625539E41AF5B37FF2CF974D6BB15CE714C98022127BDC4D208AECA6176165DB96E30407DC1E34D3F27ADD0032327A2
  (e #010001#)
  (d #050D4C44223DC46003AB35DF4F602C93A7FB338CDD6697204681DED502C4422F0FCE416302C82AF23EFD56935C264B2D559C01B379F9ADF4B057C1B7C9C2C7EA0E1A0BE5B88EB00FC3331894B2B70F3A42245B9777B39C088D32BFDC9
  (p #00D11E530BE7C0C2C416C05CB980F839330D7155F00BD92C4CA068F23F6D37DBC0908D8C270C794C8D96E94EC9451555B8997D0023CED10732558CB4EB515E8B92AA81AA68F941A9247668BE8B4D8716D57DB4B2DC15F5081286FAEA7
  (q #00FC0FC6900DD9A7C7467D0CE13C279CC91B30D3BA8ECD2172DA29AA1D2FF502EC6C5B093FBB8A26AE09B4775D1579B0CBB63FF84365E3855B23588FDBE47E3EC040C7878B239511CE7549846FB1F555B6C0BB3D4A3142E3CE18E377
  (u #102CEA514307C06E64FD857B95C90D0F80E9E5857CF1E0ABAE52250EA5AD8FC96E4A1E3B2A6E1288F4C426FCF76678F046D8DBAF8D5C69A8AD73B425763F8B9013EAB01CB6559529171333FB573C0521FD5FEABD4E2CEC609326E067
  )
 )

ciphertext:
(enc-val
 (rsa
  (a #00CAF048D7B5D6174E1FFDDA7E30A83FB0F999213EC3F9CBD7158B0E4CFBC19BA4727484BD8438BB5D27F3C2C694868FC4130461D2045AF5A9673899C3FB41314AE64D3E94780302921DA15310717227855D98685A8AE89A575721DD
  )
 )


deciphered text:
31373438363338343932373436353766616465666561626335303034373275646A646877687973777976627663766262636A616E636E636B6B736E636A736873766163616A62636261686162630A
```

5).The image below shows the signature I generate by using RSA 4096, it can be verified by libgcrypt function gcry_pk_verify().

```
RSA digital authentication signature :
(sig-val
 (rsa
  (s #A102419117DE6D7CD71D9199453F6D94FA55ED7EEA765B9BCD7AFD2E90B569D073C6B1E4C4B1068F831B23705BDED65DA8E5B46F99ECA1E9A30B982E4367F446AD3C87959D2E674A45DDD59B8B095656C2A40DAF28C216914CC111EDA
  )
 )

verify signature, if it is valid then indicating that     program is in good shape!

valid 4096 signature!
```

6).The image below is the hash value of three different HMAC and signature generated by SHA256 and RSA4096, signature can be verified by libgcrypt function gcry_pk_verify().

```
MD5 HMAC testing:
586c4485191d42d96e682b1e6aee3cdc

HMAC SHA1 testing:
a67f5e9358a6192c9543f3775cdbb42880a59eb0

HMAC SHA256 testing:
feb65d7f7650c9db4a237147d6d9121543a5ff9d25e078be7b10b56c9a2f4539

SHA256 + RSA 4096 testing:
For Digital Signature using SHA256 and RSA4096:
Algorithm: SHA256
feb65d7f7650c9db4a237147d6d9121543a5ff9d25e078be7b10b56c9a2f4539
using SHA256 to generate signature key:
(sig-val
 (rsa
  (s #8C0F985F38D07C370827381603EEAAF8B0C37C4660CB044A32EF0D194B513537545AE09C1C8BFD52E289051F16F5F680E7D67DEF3F3AFAA791EFE4AF9DFCEFC177468FFF9919939BFBF8FE052F8685192B2DB0605C02C13F40A2B3E4
  )
 )

digital signature is correct, SHA256 RSA success!
```

4 Implementation Phase Two

In phase two I conducted the 100 iteration experiments on each operation and calculated the results. The command lines are:

$tar xvfz Wang-assign2.tgz
$cd Wang-assign2
$make
$./cryptogator input_file
*****print out results******
The snapshot captured on Ubuntu Terminal is showed below:

```
result summary:
For AES128, median is 1.230000 , avg is 1.983240
For AES256, median is 1.230000 , avg is 1.259620
For RSA1024, median is  1089.753000, avg is 1138.942120
For RSA4096, median is 45122.661000 , avg is 44316.408870
For MD5, median is 3.206000 , avg is 7.836840
For SHA1, median is 0.868000 , avg is 0.882160
For SHA256, median is 3.528000 , avg is 8.569030
wangxiaohan@wangxiaohan-VirtualBox:~/Documents/Wang-assign2$
```

The ranking of performance of operations are SHA1, AES256, AES128, MD5, SHA256, RSA1024, RSA4096. It may have randomness resulted from process scheduling and context switching triggered by other running processes at that time in the machine.

5 Epilogue

The result shows that RSA consumes much more time than other operations. This may indicate that in practice, RSA is usually used as part of a hybrid cryptosystem. Rather than using the RSA key pair to encrypt and decrypt the data, cryptosystem generates a unique symmetric key, encrypts this symmetric key with RSA and at the same time encrypts the message with other faster schemes. In this way RSA is only used to encrypt a key, which is much smaller than the whole datasets.