Experiment no 5

Aim: To Implement Pretty Good Privacy (PGP) security method.

Theory:

WHAT IS PRETTY GOOD PRIVACY (PGP)?

PGP encryption or Pretty Good Privacy encryption, is a data encryption computer program that gives cryptographic privacy and authentication for online communication. It is often used to encrypt and decrypt texts, emails, and files to increase the security of emails. PGP encryption uses a mix of data compression, hashing, and public-key cryptography. It also uses symmetric and asymmetric keys to encrypt data that is transferred across networks. It combines features of private and public key cryptography. Each step uses a different algorithm, and each public key is associated with a username and an email address. PGP is used for signing, encrypting, and decrypting texts, e-mails, files, directories, and whole disk partitions and to increase the security of e-mail communications.
When plaintext is encrypted with PGP, it first compresses the plaintext. Data compression saves transmission time, disk space, and reinforces cryptographic security. Most cryptanalysis methods exploit patterns that are found in the plaintext. However, the asymmetry of PGP encryption allows for authentication. After public keys have been traded among partners, the private keys are used to digitally sign the encrypted content. This allows the decryptor to confirm the sender.

USES OF PGP ENCRYPTION

One use of PGP encryption is to confidentially send messages. To do this, PGP combines private-key and public-key encryption. The sender encrypts the message using a public encryption algorithm provided by the receiver. The receiver provides their personal public-key to whomever they would like to receive messages from. This is done to protect the message during transmission. Once the recipient receives the message, they use their own private-key to decode the message, while keeping their personal private-key a secret from outsiders.

Another aspect of PGP is message authentication and integrity checking. Integrity checking is used to detect if a message has been altered after it was written and to determine if it was actually sent by the claimed sender. Because the email is encrypted, changes in the message will make it unable to be decrypted with the key. PGP is used to create a digital signature for the message by computing a hash from the plaintext and producing a digital signature using the sender's private key. A person can add their signature to another person's public-key to show that it is truly that rightful owner.

PGP also ensures that the message belongs to the intended recipient. PGP includes requirements for distributing user's public keys in an identity certificate. These certificates are constructed so that tampering can be easily detected. The certificates can only prevent corruption after they

have been made, but not before. PGP products also help to determine if a certificate belongs to the person that is claiming it, often referred to as a web of trust.

BENEFITS OF PGP ENCRYPTION

- Sensitive information is always protected. It cannot be stolen or viewed by others on the internet. It assures that the information that is sent or received was not modified in transmission and that files were not changed without your knowledge.
- Information can be shared securely with others including groups of users and entire departments.
- You can be certain who the email is from and who it is for. PGP verifies the sender of the information to ensure that the email was not intercepted by a third party.
- Your secure emails and messages cannot be penetrated by hackers or infected by email attacks.
- Others cannot recover sensitive messages or files once you have deleted them.
- PGP encryption software is very easy to learn how to use. With virtually no training, users are able to learn how to use it right away.

PGP encryption uses a serial combination of hashing, data compression, symmetric-key cryptography, and finally public-key cryptography; each step uses one of several supported algorithms. Each public key is bound to a username or an e-mail address. PGP consists of the following five services (refer fig 1):

1. Authentication
2. Confidentiality
3. Compression
4. E-mail compatibility
5. Segmentation

Function	Algorithms Used	Description
Digital signature	DSS/SHA or RSA/SHA	A hash code of a message is created using SHA-1. This message digest is encrypted using DSS or RSA with the sender's private key and included with the message.
Message encryption	CAST or IDEA or Three-key Triple DES with Diffie-Hellman or RSA	A message is encrypted using CAST-128 or IDEA or 3DES with a one-time session key generated by the sender. The session key is encrypted using Diffie-Hellman or RSA with the recipient's public key and included with the message.
Compression	ZIP	A message may be compressed, for storage or transmission, using ZIP.
Email compatibility	Radix 64 conversion	To provide transparency for email applications, an encrypted message may be converted to an ASCII string using radix 64 conversion.
Segmentation	—	To accommodate maximum message size limitations, PGP performs segmentation and reassembly.

Fig 1: PGP Services

Authentication:

Figure 2 illustrates the digital signature service provided by PGP. The figure is similar to ones we have looked at earlier. The hash function used is SHA-1 which creates a 160 bit message digest. EP (DP) represents public encryption (decryption) and the algorithm used can be RSA or DSS (recall that the DSS can only be used for the digital signature function and unlike RSA cannot be used for encryption or key exchange). The message may be compressed using and algorithm called ZIP. This is represented by "Z" in the figure (refer Fig 2).

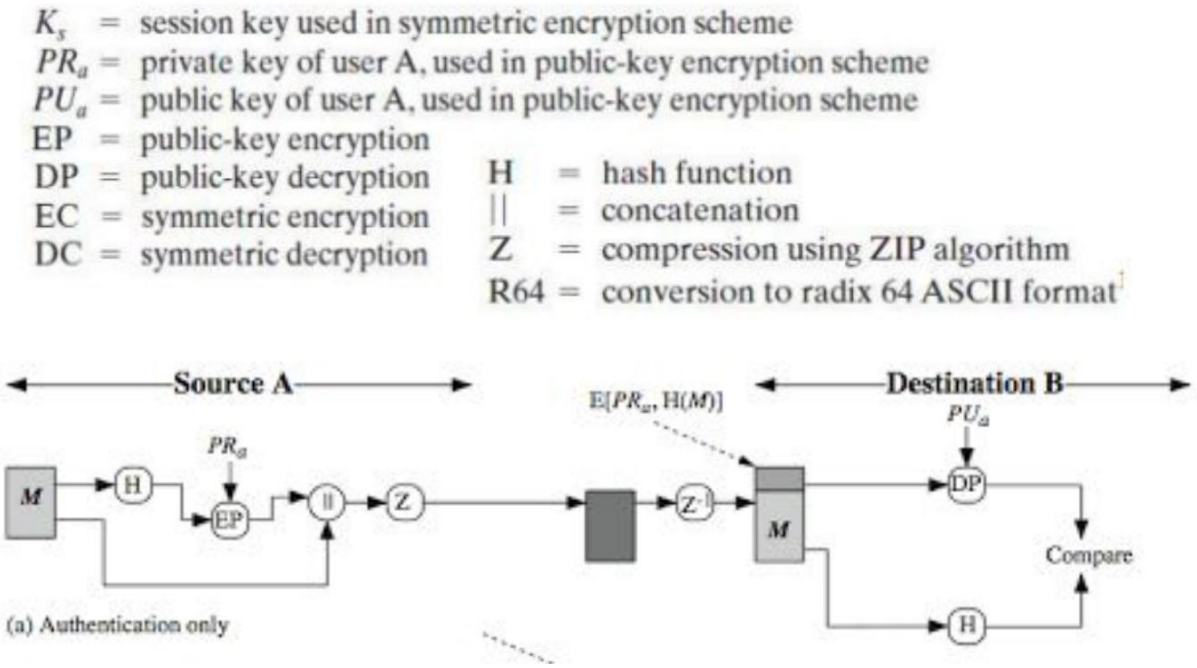

Fig 2: Authentication

The combination of SHA-1 and RSA provides an effective digital signature scheme. Due to the strength of RSA the recipient is assured that only the possessor of the matching private key can generate the signature. Because of the strength of SHA-1 the recipient is assured that no one else could generate a new message that matches the hash code and hence, the signature of the original message.

Confidentiality:

Another basic service provided by PGP is confidentiality which is provided by encrypting messages to be transmitted or to be stored locally as files. In both cases, the user has a choice of CAST-128, IDEA or 3DES in 64 bit cipher feedback (CFB) mode. The symmetric key is used only once and is created as a random number with the required number of bits. It is transmitted along with the message and is encrypted using the recipients' public key. Figure 3 illustrates the sequence:

1. The sender generates a message and a random number to be used as a session key for this message only.
2. The Message is encrypted using CAST-128, IDEA or 3DES with the session key.
3. The session key is encrypted with RSA (or another algorithm known as ElGamal) using the recipients public key and is prepended to the message.

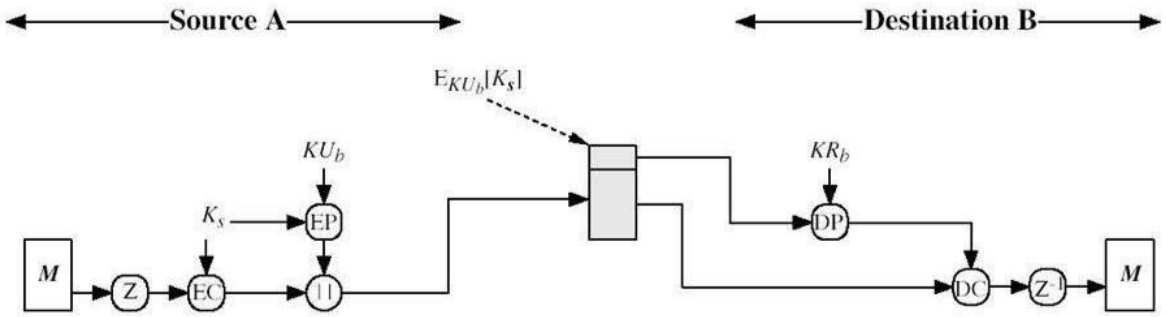

Fig 3: Confidentiality

Authentication & Confidentiality:

As figure 4 illustrates, both services may be used for the same message. First, a signature is generated for the plaintext message and prepended to the message. Then the plaintext message plus signature is encrypted using CAST-128 (or IDEA or 3DES), and the session key is encrypted using RSA (or ElGamal). This sequence is preferable to the opposite: encrypting the message and then generating a signature of the encrypted message. It is generally more convenient to store a signature with a plaintext version of a message. Furthermore, for purposes of third party verification, if the signature is performed first, a third party need not be concerned with the symmetric key when verifying the signature.

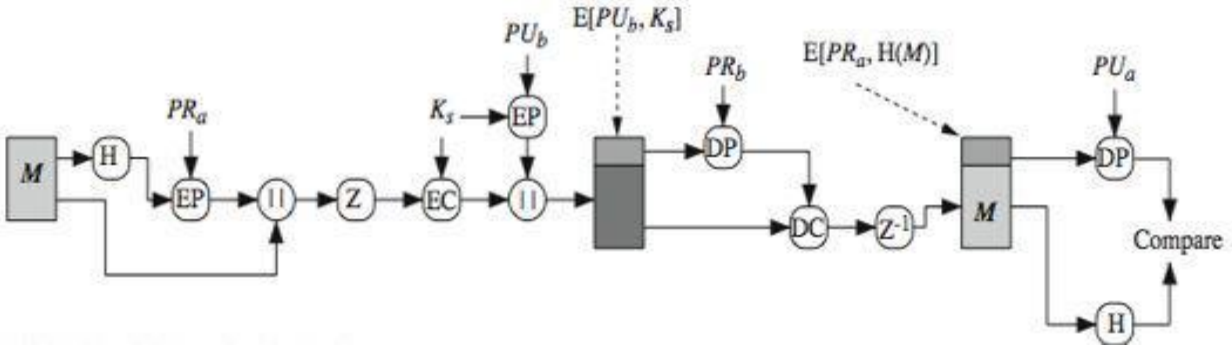

Fig 4: Authentication & Confidentiality

Compression:

As a default, PGP compresses the message after applying the signature but before encryption. This has the benefit of saving space both for e-mail transmission and for file storage. The

placement of the compression algorithm, indicated by Z for compression and Z^{-1} for decompression in figure 4 is critical:

1. The signature is generated before compression for two reasons:
   (a) It is preferable to sign an uncompressed message so it is free of the need for a compression algorithm for later verification.
   (b) Different version of PGP produces different compressed forms. Applying the hash function and signature after compression would constrain all PGP implementation to the same version of the compression algorithm.
2. Message encryption is applied after compression to strengthen cryptographic security. Because the compressed message has less redundancy than the original plaintext, cryptanalysis is more difficult.

Run-length encoding (**RLE**)

Run-length encoding (RLE) is a very simple form of lossless data compression in which runs of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. This is most useful on data that contains many such runs. Consider, for example, simple graphic images such as icons, line drawings, Conway's Game of Life, and animations. It is not useful with files that don't have many runs as it could greatly increase the file size.

RLE may also be used to refer to an early graphics file format supported by CompuServe for compressing black and white images, but was widely supplanted by their later Graphics Interchange Format. RLE also refers to a little-used image format in Windows 3.x, with the extension rle, which is a Run Length Encoded Bitmap, used to compress the Windows 3.x startup screen.

For example, if the input string is "wwwwaaadexxxxxx", then the function should return "w4a3d1e1x6".

a) Pick the first character from source string.

b) Append the picked character to the destination string.

c) Count the number of subsequent occurrences of the picked character and append the count to destination string.

d) Pick the next character and repeat steps b) c) and d) if end of string is NOT reached

Algorithm:

1. Sender creates a message M
2. Message is hashed using SHA-1 H(M)
3. Encrypting H(M) using RSA with senders' private key EP (PRa ,H(M))
4. Appending Encrypted hashed to Message M = M + EP(PRa , H(M))

5. Zipping the appended with Run-length encoding algorithm Z(M)
6. Generating random session key K and encrypting the message EC(K, M) using AES
7. Encrypting the session key also using RSA with Receivers' public key EP(PUb, K)
8. At Receivers' End, Message is decrypted using receivers' private key to get the session key K using RSA DP(PRb, M)
9. Session key is used to decrypt the received the message using AES decryption DC(K,M)
10. Decrypted message is unzipped Z^{-1}
11. Unzipped message is decrypted using public key of sender DC(PUa, M)
12. Message is hashed with SHA-1 to get H(M)
13. Both the values are compared DC(PUa, M) and H(M) and if both are equal , confidentiality and authentication is achieved.

Code:

```
import hashlib
import rsa
from Crypto import Random
from Crypto.PublicKey import RSA
import zlib
import base64
import pyDes
import itertools
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

def compress(string):
    n = len(string)
    i=0
    result=""
    while i < n :
      count = 1
      while i < n - 1 and string[i] == string[i + 1]:
          count=count+1
          i=i+1
      result=result+string[i]+str(count)
      i=i+1
    return result

def decompress(string):
  result=""
  n=len(string)
  i=0
  while(i<=(n-2)):
    c=string[i]
    times=int(string[i+1])
```

```
    result=result+c*times
    i=i+2
  return result


def decode_base64_and_inflate( b64string ):
    decoded_data = base64.b64decode( b64string )
    return zlib.decompress( decoded_data , -15)

def deflate_and_base64_encode( string_val ):
    zlibbed_str = zlib.compress( string_val )
    compressed_string = zlibbed_str[2:-4]
    return base64.b64encode( compressed_string )

message=input("Enter your message : ")
t=len(message)
# encoding message using encode()
# then sending to SHA1()
hashvalue = hashlib.sha1(message.encode())
# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA1 is : ")
print(hashvalue.hexdigest())
(pubkeyA, privkeyA) = rsa.newkeys(512)

cypher_hashcode = rsa.encrypt(hashvalue.hexdigest().encode('utf-8'),privkeyA)
print("Encrypted Hashcode is:\n",cypher_hashcode)
print("Hashcode size ",len(cypher_hashcode))
message = message+str(cypher_hashcode)
print("Message + Encrypted Hashcode is:\n",message)
#print(message.encode('utf-8'))
#message=message.encode('utf-8')
#messageb64=base64.b64encode(message)
#print(messageb64)
print("Compressing the string: ")
compress_string=compress(message)
print(compress_string)

aes_key = b'Sixteen byte key'
print("AES key is :")
print(aes_key)
iv = Random.new().read(AES.block_size)
print(len(iv))
cipher = AES.new(aes_key, AES.MODE_CFB, iv)
zip_encryption = iv + cipher.encrypt(compress_string)
print("Encryption of Zip ")
print(zip_encryption)
```

```
(pubkeyB, privkeyB) = rsa.newkeys(512)
key_encryption=rsa.encrypt(aes_key,pubkeyB)
print("Encryption of Key")
print(key_encryption)
print(len(key_encryption))
final_code=zip_encryption+key_encryption
print("Final Msg")
print(final_code)
print("--------------------------------------------------")
print("Decryption of Key")
key_decryption=rsa.decrypt(final_code[-64:],privkeyB)
print(key_decryption)
print("Decryption of Zip :")
zip_decryption=cipher.decrypt(final_code[:-64])
zip_decryption=zip_decryption[16:]
print(zip_decryption)
zip_decryption=zip_decryption.decode('utf8')
print(zip_decryption)
decompress_zip=decompress(zip_decryption)
print(decompress_zip)
r_cypher_hashcode=decompress_zip[t:]
print(r_cypher_hashcode)
decrypted_hashcode=rsa.decrypt(r_cypher_hashcode,pubkeyA)
print(decrypted_hashcode)
received_message=decompress_zip[:t]
print(“Received message”)
print(received_message)
hashvalue_r = hashlib.sha1(received_message.encode())
print("The hexadecimal equivalent of SHA1 of received message : ")
print(hashvalue_r.hexdigest())
if(hashvalue_r.hexdigest()==hashvalue.hexdigest()):
  print("message is untouched")
else:
  print("message is touched")
```

Output:

```
# Python 3.6.1 (default, Dec 2015, 13:05:11)
# [GCC 4.8.2] on linux
# Enter your message : hibuddyyyyy
# The hexadecimal equivalent of SHA1 is :
# 87e41daf6e4388ebe24ba97c22b3679f8468fdce

# Encrypted Hashcode is:
b'\xe7l\\\x9e\x9ak\xa2\xec(\xdaW\xce\xf8\xd87\xd1\xd10\x9f\xb27\xdd\xbf*&\x93\xcd\x8exn\x8dG#\xf8\xad\xd5\xa6wRX>\xa2\x98\x1f\xcdJ\x13>\xb2\xbfw"H\xbf:U@[\x90\xee\xdcz\xc0\xa2'

# Hashcode size  64
# Message + Encrypted Hashcode is:
#hibuddyyyyyb'\xe7l\\\x9e\x9ak\xa2\xec(\xdaW\xce\xf8\xd87\xd1\xd10\x9f\xb27\xdd\xbf*&\x93\xcd\x8exn\x8dG#\xf8\xad\xd5\xa6wRX>\xa2\x98\x1f\xcdJ\x13>\xb2\xbfw"H\xbf:U@[\x90\xee\xdcz\xc0\xa2'

# Compressing the string:
#h1i1b1u1d2y5b1'1\1x1e171l1\3x191e1\1x191a1k1\1x1a121\1x1e1c1(1\1x1d1a1W1\1x1c1e1\1x1f181\1x1d18171\1x1d111\1x1d11101\1x191f1\1x1b12171\1x1d2\1x1b1f1*1&1\1x19131\1x1c1d1\1x181e1x1n1\1x181d1G1#1\1x1f181\1x1a1d1\1x1d151\1x1a161w1R1X1>1\1x1a121\1x19181\1x111f1\1x1c1d1J1\1x11131>1\1x1b121\1x1b1f1w1"1H1\1x1b1f1:1U1@1[1\1x19101\1x1e2\1x1d1c1z1\1x1c101\1x1a121'1
# AES key is :
# b'Sixteen byte key'
# 16
# Encryption of Zip
#
b'\xc7<\xee\x08\x01RY\xc4z\x8e\xee\xd9\xa6\xd5\xd6\xf0\xa9fP@\x1e\xbb\x94rWY`A\xee\x1e\xee\x1e<7Bv\xa0\x13j\xa9\xf9\x07\xbd\xfd\xa8\x9c\xe9\xc8\xde\xe46\xe7U[\xa3\\r\x910U"\xdc\x8c\xacGH\xe2\xf4\x05@\x1c\x16\xcc\x08S$\xb6\xb7\x96#k\xf90\x9f\x18\x87\xb9G\xb5\xff\xd0\x01\x9b\xe5\xdb\xb3\xf7\xfb\xd4\\\xd5\x03\xcaR\x11\xf0\x8b\x03\xb8\xfb\x16$]\xf5\x92o\xb8;%\xa1\x1d\xec!\x8a\x8b\xab\xc5S\x9d\xe2\x12\xd7\x9b\xa0\x07\xc9\x01\xe9\xb7\xfd\xea[\x1e\\\xdcog\x9d\x81\x8f\xb8\x8fR"\x12\xed}\xae\x03\xe1K|\xf5\xec\x8d.\xc5\xe0\x9d\xf9\x9dk3\xe2\x81\x1e\x0f\xa5\xcf&\xfa\x8fM\xbf\x04\x14\x85\x0b\xa1\xa8R\x90(\t\x01\x8f\xd9\xd0\xf2\x87\x00="\xa2\x10\xfd7\xfb\x95\xdaU+pE\xa6\xac\x1ch\x1c!Gl\x91q\xc8V\x17A\x13{\xe9\xf1\xfa\xbf\x9b\x0f\xe25\x87\xfe\xf2\x10\x9bJJ\x07\xcd\x18\x9c\x11\xb7\xec\xa2;\x9d^P\x1dSW\xb8\x90(\xaa>\xed\xa9\xa0\xc6\xb3\rb[\xe7\xca\xf9\x03\xa4BZ\x91c\ri\xad7^\x8e^#WW\xc3\xa6\xe9\x9e\x11mQ\x02}>\xbf\xa5)\x93\xfcd\xcd\xec@l\xf4Rx\x9ea\x00\xcd\xca\xb5\xd6\x9coji\xcd4\xbe\x92\x17\xc9\xcc\xcf\xd5\x0f\xe1G\xc7kO!\xddEA;.r\xb48\x99/Q\xec\x8c\x7f\xa4\xd9\x93S\xf2*\x86\x9a^\x85\xbc\xadR\xdc\xa6'
# Encryption of Key
#
b'I\xb5\xc7b\xf6\x01I\xb8N\x01\xd4r\xfb,\xee\x01q\xdf\x81\xc9V\x85\xe9\xd4\x9f\xeb\xfdF\x8
```

1\x94!\x80+\xea\xdb9\xa0\x15T\x06\x05\x12\x19\x0f\xcb\xfd\xba!YeH\xf6Hm\xe5\xf6\xbd\xab
\xd3i\x9aJ\x1b5'

Key Size : 64

Final Msg
#
b'\xc7<\xee\x08\x01RY\xc4z\x8e\xee\xd9\xa6\xd5\xd6\xf0\xa9fP@\x1e\xbb\x94rWY`A\xee\x1
e\xee\x1e<7Bv\xa0\x13j\xa9\xf9\x07\xbd\xfd\xa8\x9c\xe9\xc8\xde\xe46\xe7U[\xa3\\r\x910U"\x
dc\x8c\xacGH\xe2\xf4\x05@\x1c\x16\xcc\x08S$\xb6\xb7\x96#k\xf90\x9f\x18\x87\xb9G\xb5\xf
f\xd0\x01\x9b\xe5\xdb\xb3\xf7\xfb\xd4\\\xd5\x03\xcaR\x11\xf0\x8b\x03\xb8\xfb\x16$]\xf5\x92
o\xb8;%\xa1\x1d\xec!\x8a\x8b\xab\xc5S\x9d\xe2\x12\xd7\x9b\xa0\x07\xc9\x01\xe9\xb7\xfd\xea
[\x1e\\\xdcog\x9d\x81\x8f\xb8\x8fR"\x12\xed}\xae\x03\xe1K|\xf5\xec\x8d.\xc5\xe0\x9d\xf9\x9d
k3\xe2\x81\x1e\x0f\xa5\xcf&\xfa\x8fM\xbf\x04\x14\x85\x0b\xa1\xa8R\x90(\t\x01\x8f\xd9\xd0\
xf2\x87\x00="\xa2\x10\xfd7\xfb\x95\xdaU+pE\xa6\xac\x1ch\x1c!Gl\x91q\xc8V\x17A\x13{\xe9
\xf1\xfa\xbf\x9b\x0f\xe25\x87\xfe\xf2\x10\x9bJJ\x07\xcd\x18\x9c\x11\xb7\xec\xa2;\x9d^P\x1d
SW\xb8\x90(\xaa>\xed\xa9\xa0\xc6\xb3\rb[\xe7\xca\xf9\x03\xa4BZ\x91c\ri\xad7^\x8e^#WW\x
c3\xa6\xe9\x9e\x11mQ\x02}>\xbf\xa5)\x93\xfcd\xcd\xec@l\xf4R
x\x9ea\x00\xcd\xca\xb5\xd6\x9coji\xcd4\xbe\x92\x17\xc9\xcc\xcf\xd5\x0f\xe1G\xc7kO!\xddEA
;.r\xb48\x99/Q\xec\x8c\x7f\xa4\xd9\x93S\xf2*\x86\x9a^\x85\xbc\xadR\xdc\xa6I\xb5\xc7b\xf6\x
01I\xb8N\x01\xd4r\xfb,\xee\x01q\xdf\x81\xc9V\x85\xe9\xd4\x9f\xeb\xfdF\x81\x94!\x80+\xea\x
db9\xa0\x15T\x06\x05\x12\x19\x0f\xcb\xfd\xba!YeH\xf6Hm\xe5\xf6\xbd\xab\xd3i\x9aJ\x1b5'
--
Decryption of Key
b'Sixteen byte key'

Decryption of Zip :
#b'h1i1b1u1d2y5b1\'1\\1x1e171l1\\3x191e1\\1x191a1k1\\1x1a121\\1x1e1c1(1\\1x1d1a1W1\\1x1
c1e1\\1x1f181\\1x1d18171\\1x1d111\\1x1d11101\\1x191f1\\1x1b12171\\1x1d2\\1x1b1f1*1&1\\
1x19131\\1x1c1d1\\1x181e1x1n1\\1x181d1G1#1\\1x1f181\\1x1a1d1\\1x1d151\\1x1a161w1R1X
1>1\\1x1a121\\1x19181\\1x111f1\\1x1c1d1J1\\1x11131>1\\1x1b121\\1x1b1f1w1"1H1\\1x1b1f1:
1U1@1[1\\1x19101\\1x1e2\\1x1d1c1z1\\1x1c101\\1x1a121\'1'
#
h1i1b1u1d2y5b1'1\1x1e171l1\3x191e1\1x191a1k1\1x1a121\1x1e1c1(1\1x1d1a1W1\1x1c1e1\1x
1f181\1x1d18171\1x1d111\1x1d11101\1x191f1\1x1b12171\1x1d2\1x1b1f1*1&1\1x19131\1x1c
1d1\1x181e1x1n1\1x181d1G1#1\1x1f181\1x1a1d1\1x1d151\1x1a161w1R1X1>1\1x1a121\1x19
181\1x111f1\1x1c1d1J1\1x11131>1\1x1b121\1x1b1f1w1"1H1\1x1b1f1:1U1@1[1\1x19101\1x1
e2\1x1d1c1z1\1x1c101\1x1a121'1
#
hibuddyyyyyb'\xe7l\\\x9e\x9ak\xa2\xec(\xdaW\xce\xf8\xd87\xd1\xd10\x9f\xb27\xdd\xbf*&\x93
\xcd\x8exn\x8dG#\xf8\xad\xd5\xa6wRX>\xa2\x98\x1f\xcdJ\x13>\xb2\xbfw"H\xbf:U@[\x90\xe
e\xdcz\xc0\xa2'
#
b'\xe7l\\\x9e\x9ak\xa2\xec(\xdaW\xce\xf8\xd87\xd1\xd10\x9f\xb27\xdd\xbf*&\x93\xcd\x8exn\x
8dG#\xf8\xad\xd5\xa6wRX>\xa2\x98\x1f\xcdJ\x13>\xb2\xbfw"H\xbf:U@[\x90\xee\xdcz\xc0\xa
2'

#87e41daf6e4388ebe24ba97c22b3679f8468fdce
#Received message
#Hibuddyyyyy
#The hexadecimal equivalent of SHA1 of received message :
#87e41daf6e4388ebe24ba97c22b3679f8468fdce
message is untouched

Conclusion:

Authentication and Confidentiality for the message is achieved using PGP Security System. We can use this same system in our email application. We also saw the various pros and cons of pretty good privacy and we thus conclude that pretty good privacy is a great technique or system in cryptography.