Week 9 Homework due on November 16, 18:59 Hour

Group 5

Wong Ann Yi (1004000)
Liu Bowen (1004028)
Tan Chin Leong Leonard (1004041)

Exercise 1

Design and implement a secure channel. Use the following interface:

```
class Peer(object):
        def __init__(self, key):
                ...
        def send(self, msg):
                ... # protect the message
                return protected_msg # type of protected_msg is 'str'
        def receive(self, protected_msg):
                ... # verify the message and print errors if any
                print msg # successfuly recovered plaintext
# Example
alice = Peer("very secret key!")
bob = Peer("very secret key!")

msg1 = alice.send("Msg from alice to bob")
bob.receive(msg1)

msg2 = alice.send("Another msg from alice to bob")
bob.receive(msg2)

msg3 = bob.send("Hello alice")
alice.receive(msg3)
```

Answers:

To implement a secure channel, we need a shared secret key. In this case we will assume that Alice and Bob share a secret key *K*, but that nobody else knows this key. The key *K* is known only to Alice and Bob and derived from the initialization key by using the HKDF() in python.

Crypto.Protocol.KDF.HKDF(master, key_len, salt, hashmod, num_keys=1, context=None) can derive one or more keys from a master secret using the HMAC-based KDF.

The secure channel is designed to achieve a security level of 128 bits. Following Chpater 3.5.7, we will use a 256-bit key. Thus, key_len = 32 bytes.

There are theoretical results that show that, given certain specific definitions of secure encryption and authentication, the encrypt-first solution is secure. It is also more efficient.

In the initialize step, we generate the two keys derived from the master key ”very secret key!”: one is for key for encryption and another is for HMAC. Using different keys for both processes increases the security.

Next, we need to set two counters and one is for tagging the number of sending messages and another is for tagging the number of receiving messages.

When sending the message, we utilize the AES CBC mode to encrypt the message where the IV is generated by first 16bytes of sha256(sender_counter). As the sender_counter increases with each sending process, the IV is dynamic and is more secure. The sending message size may not be in whole number of the multiple block sizes. Therefore, padding will be implemented to ensure complete block size. After encryption, we will generate the authentication tag by using HMAC where the hash function is hash256.

After encryption and authentication, we send the whole message which consists of (sender_counter || ciphertext || HMAC) to the receiver and then increase the sender_counter = sender_counter + 1.

When the receiver notices the sending message, he first parses the whole message and segment it into the counter = first 32-bit of message, the authentext (or tag) = the last 128-bit of message and the ciphertext which is in the middle of the message body.

To authenticate the message, the receiver will generate the tag using hash256(ciphertext) and compare it hash256(ciphertext) == authentext. If both tags are the same, the receiver will decrypt the ciphertext otherwise he will discard the whole message.

The codes in Python 2.7 is shown in:

```
from Cryptodome.Protocol import KDF
from Crypto.Hash import SHA256
from Crypto.Cipher import AES
import binascii
import hashlib
import hmac
class Peer(object):
    def __init__(self, key):
        self.share_key = key;
        #set counter for sending and receiving
```

```
        self.sender_counter = 0;
        self.receiver_counter = 0;
        #derive the encryption key and authentication key from share_key
        self.encry_key = (KDF.HKDF(self.share_key, salt=None, key_len=32,
hashmod=SHA256, num_keys=2, context=None))[0]
        self.auth_key = (KDF.HKDF(self.share_key, salt=None, key_len=32,
hashmod=SHA256, num_keys=2, context=None))[1]

    def send(self, msg):
        #generate a new IV based on current
        #sender_counter for each sending process
        IV = hashlib.sha256(format(self.sender_counter, 'x')).hexdigest()[:16]
        cipher = AES.new(self.encry_key, AES.MODE_CBC, IV)
        #padding if needed
        #use '0x80' + rest is '0x00' padding mode
        if len(msg) % 16 == 0:
            message = msg
        else:
            message = msg.encode('hex') + '80' + (16 - len(msg) % 16 - 1)*'00'
        ciphertext = binascii.hexlify(cipher.encrypt(message))
        authtext = (hmac.new(self.auth_key, ciphertext, hashlib.sha256).hexdigest())
        counter_len = len((format(self.sender_counter, 'x')))
        #set counter to 32-bit(4 bytes)
        #each format() has 4-bit so 32-bit == (4*2) * 4-bit
        if counter_len < 4*2:
            counter = (4*2 - counter_len)*'0' + (format(self.sender_counter, 'x'))
        else:
            counter = counter_len
        #sending message is counter || ciphertext || authtext
        protected_msg = counter + ciphertext + authtext
        #increase the sender_counter
        self.sender_counter += 1
        print 'whole sending message is:', protected_msg
```

```
        return protected_msg


    def receive(self, protected_msg):
        size = len(protected_msg)
        #extract authtext (last 256-bit)
        authtext =  protected_msg[size-64:]
        #extract ciphertext (middle part)
        ciphertext = (protected_msg[8:size-64])
        #extract counter (first 32-bit)
        counter = int(protected_msg[0:8], 16)
        #first verify the authtext
        verify_authtext = hmac.new(self.auth_key, (ciphertext),
hashlib.sha256).hexdigest()
        if (authtext == verify_authtext):
            #generate the IV which is same as the sender
            IV = hashlib.sha256(format(counter, 'x')).hexdigest()[:16]
            cipher = AES.new(self.encry_key, AES.MODE_CBC, IV)
            msg = cipher.decrypt((ciphertext).decode('hex'))
            #remove the padding
            msg_size = len(msg)
            for i in range(msg_size / 2):
                if msg[msg_size-2:] == '00':
                    msg = msg[0:msg_size-2]
                    msg_size -= 2
                    continue;
                if msg[msg_size-2:] == '80':
                    msg = msg[0:msg_size-2]
                    msg_size -= 2
                    break;
            #increase the receiver_counter
            self.receiver_counter += 1
            #print the native message
```

```
        print 'message to receiver is:', binascii.unhexlify(msg)
    else:
        print 'authenticate failed!'

alice = Peer("very secret key!")
bob = Peer("very secret key!")
msg1 = alice.send("Msg from alice to bob")
bob.receive(msg1)
msg2 = alice.send("Another msg from alice to bob")
bob.receive(msg2)
msg3 = bob.send("Hello alice")
alice.receive(msg3)
```

The msg1 result is shown in:

00000000eb5dea4757933bea0cdec258ebeb56798f756e55718c47693bf6159395ea82cf9312fd686
dba5e1e51a8ccc628ba688e8f64ee37fdb68ca7a4f5c9b69970ecd5522e7ec2ceab5914d856eabb4b
9045ec80be57e2c3ce1597728721007e2508f3

The black text is the sender counter, blue text is the ciphertext and red text is the authenticate tag.

The receiver parses the message and obtains:

```
message to receiver is: Msg from alice to bob
```

The msg2 result is shown in:

000000015797ee793de3df31783ec308fd7d83347993d26b13a59b4298ab3afe1aa9f897e3d2ee62
bf6ea9ede4e781812d662a551783086f1429748331e3502b8ce0c58a007686c7e2b4907588005ff3
355b1abeac3ec5d85e71a2d4a3528958586ba310

The black text is the sender counter, blue text is the ciphertext and red text is the authenticate tag.

The receiver parses the message and obtains:

```
message to receiver is: Another msg from alice to bob
```

The msg3 result is shown in:

00000000148312496f04e9a75cfb88f3955fc66a38fcca14cb5fc2cc515390e3b306d4067f89cbd08
8470423be72718c6cc210d2be7adc87f89459ca0eccaeec268df828

The black text is the sender counter, blue text is the ciphertext and red text is the authenticate info.

The receiver parses the message and obtains:

```
message to receiver is: Hello alice
```

Exercise 2

Describe how SSL/TLS protect confidentiality and integrity of messages.

Answers:

Confidentiality

SSL and TLS use a combination of symmetric and asymmetric encryption to ensure message privacy. During the SSL or TLS handshake, the SSL or TLS client and server agree an encryption algorithm and a shared secret key to be used for one session only. All messages transmitted between the SSL or TLS client and server are encrypted using that algorithm and key, ensuring that the message remains private even if it is intercepted. SSL supports a wide range of cryptographic algorithms. Because SSL and TLS use asymmetric encryption when transporting the shared secret key, there is no key distribution problem. The recommended widely used symmetric key is AES and the asymmetric key is RSA.

The below figures show how symmetric and asymmetric encryption works[1].

Figure 1. Symmetric key cryptography

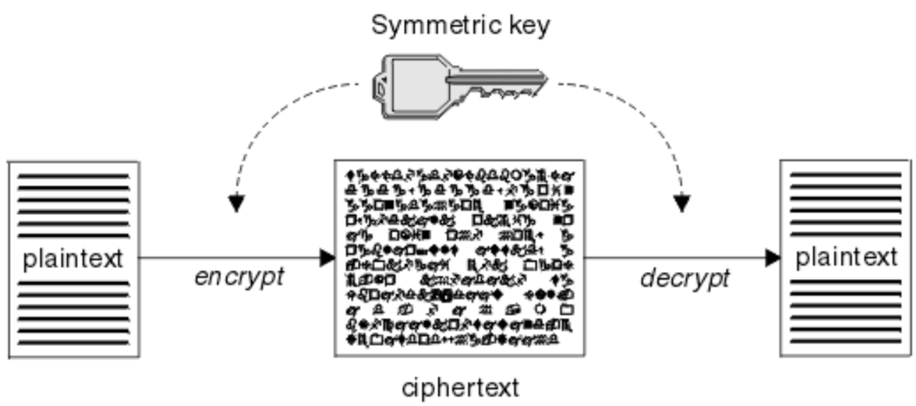
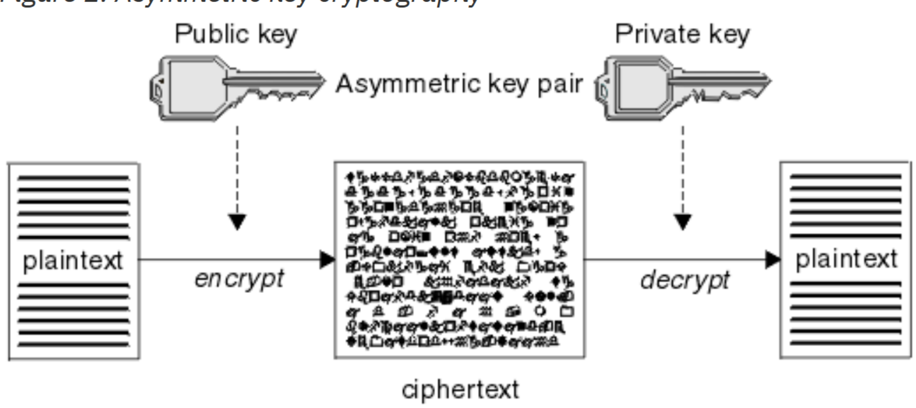


Figure 2. Asymmetric key cryptography



[1] www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.5.0/com.ibm.mq.sec.doc/q009940_.htm

Data integrity

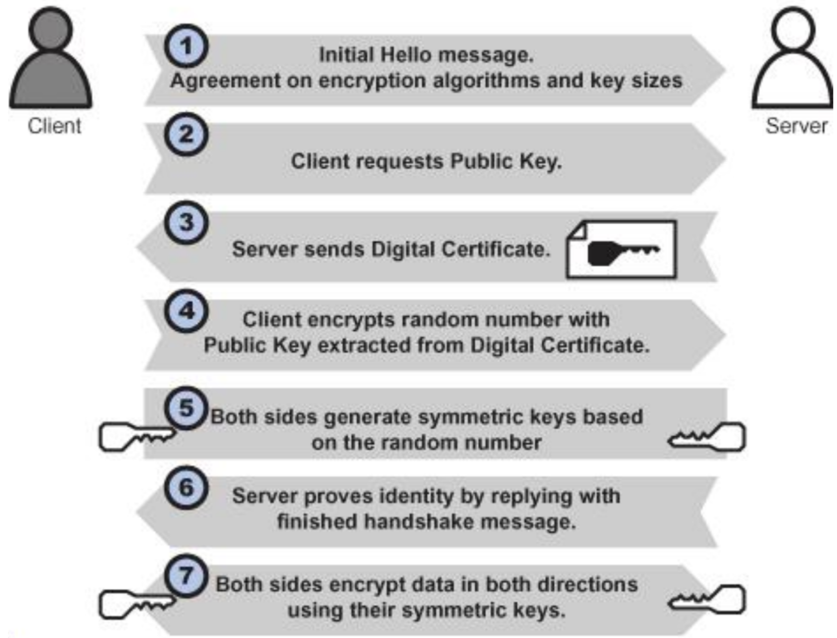
SSL and TLS provide data integrity by calculating a message digest. Use of SSL or TLS does ensure data integrity, provided that the Cipher Spec in your channel definition uses a hash algorithm as described in the table below. TLS_RSA_WITH_AES_256_CBC_SHA or TLS_RSA_WITH_AES_256_GCM_SHA384 is recommended whereby MD5 is strongly discouraged as this is now very old and is no longer secure for most practical purposes

Below is a sample of a Cipher Spec Table:

CipherSpec name	Protocol used	Data integrity	Encryption algorithm	Encryption bits	FIPS[1]	Suite B 128 bit	Suite B 192 bit
NULL_MD5 [a]	SSL 3.0	MD5	None	0	No	No	No
NULL_SHA [a]	SSL 3.0	SHA-1	None	0	No	No	No
RC4_MD5_EXPORT [2 a]	SSL 3.0	MD5	RC4	40	No	No	No
RC4_MD5_US [a]	SSL 3.0	MD5	RC4	128	No	No	No
RC4_SHA_US [a]	SSL 3.0	SHA-1	RC4	128	No	No	No
RC2_MD5_EXPORT [2 a]	SSL 3.0	MD5	RC2	40	No	No	No
DES_SHA_EXPORT [2 a]	SSL 3.0	SHA-1	DES	56	No	No	No
RC4_56_SHA_EXPORT1024 [3 b]	SSL 3.0	SHA-1	RC4	56	No	No	No
DES_SHA_EXPORT1024 [3 b]	SSL 3.0	SHA-1	DES	56	No	No	No
TLS_RSA_WITH_AES_128_CBC_SHA [a]	TLS 1.0	SHA-1	AES	128	Yes	No	No
TLS_RSA_WITH_AES_256_CBC_SHA [4 a]	TLS 1.0	SHA-1	AES	256	Yes	No	No
TLS_RSA_WITH_DES_CBC_SHA [a]	TLS 1.0	SHA-1	DES	56	No[5]	No	No
FIPS_WITH_DES_CBC_SHA [b]	SSL 3.0	SHA-1	DES	56	No[6]	No	No
TLS_RSA_WITH_AES_128_GCM_SHA256 [b]	TLS 1.2	AEAD AES-128 GCM	AES	128	Yes	No	No
TLS_RSA_WITH_AES_256_GCM_SHA384 [b]	TLS 1.2	AEAD AES-256 GCM	AES	256	Yes	No	No
TLS_RSA_WITH_AES_128_CBC_SHA256 [b]	TLS 1.2	SHA-256	AES	128	Yes	No	No
TLS_RSA_WITH_AES_256_CBC_SHA256 [b]	TLS 1.2	SHA-256	AES	256	Yes	No	No

The below figure shows the flow of the SSL/TLS Authenticate-then-Encrypt method[2].

[2] www.infosectoday.com/Articles/Intro_to_Cryptography/Introduction_Encryption_Algorithms.htm


Client
Server
1
Initial Hello message.
Agreement on encryption algorithms and key sizes
2
Client requests Public Key.
3
Server sends Digital Certificate.
4
Client encrypts random number with
Public Key extracted from Digital Certificate.
5
Both sides generate symmetric keys based
on the random number
6
Server proves identity by replying with
finished handshake message.
7
Both sides encrypt data in both directions
using their symmetric keys.

Exercise 3

Compare the advantages and disadvantages of using a PRNG vs a RNG.

Answers:

Advantages of using PRNG

PRNGs (Pseudo Random Number Generators), which are deterministic *(same inputs result in same outputs through* a given starting condition or initial state*)* random number generators, generate numbers with fast, easy, inexpensive, and hardware independent solutions. The statistical qualities of these numbers produced are close to the ideal. PRNGs must meet the requirements specified in Table 1 below (R1 to R4) to be used especially for authentication and key generation. Therefore, nondeterministic functions are added to the output functions of PRNGs to guarantee these requirements.

Disadvantages of using PRNG

If you use a PRNG, the protocol is only secure as long as the attacker cannot break the PRNG; the protocol is computationally secure. As it is deterministic, one can predict the pseudorandom value if one knows the seed. Cryptographic protocols use computational assumptions for almost everything[3]. Removing the computational assumption for one particular type of attack is an insignificant improvement, and generating real random data, which you need for the unconditional security, is so difficult that you are far more likely to reduce the system security by trying to use real random data. Any weakness in the real random generator immediately leads to a loss of security. Another problem arises if the same prng state is used more than once. This can happen when two or more virtual machines (VMs) are booted from the same state and read the same seed file from disk.

Advantages of using RNG refer to as True Random Number Generation

TRNGs (True Random Number Generators), which are nondeterministic *(same inputs but result in different outputs through* a given starting condition or initial state*)* random number generators. Contrary to PRNGs, there is no need to include extra components in the TRNG system designs for R2, R3, and R4 requirements (see Table 1). Because of the unpredictability of random numbers generated by the use of high noise sources with high entropy in TRNGs, it is assumed that the R2 requirement is met. If the R2 requirement is satisfied, then it is assumed that the R3 and R4 requirements are also satisfied. To meet the R1 requirement in TRNGs, postprocessing techniques are applied to the random numbers obtained by sampling from noise sources. This eliminates the statistical weaknesses of random numbers at the output of the TRNG. In addition, postprocessing techniques eliminate potential weaknesses and make TRNG designs strong and flexible.

[3] *Cryptography Engineering-Niels Ferguson Bruce Schneier Tadayoshi Kohno*

Table 1: Requirements for random numbers[4].	
Requirement	Explanation
R1	RNGs must generate random numbers having good statistical properties at the output to be used in cryptographic applications.
R2	In case of the attacker knows the sub-generators of random numbers, it must not be allowed to calculate or predict premise and consecutive random numbers with high accuracy.
R3	It must not be possible to predict or calculate previously generated random numbers with high accuracy by considering the known current internal state value of a RNG or without requiring its internal state information.
R4	It must not be possible to predict or calculate subsequent random numbers with high accuracy by considering the known current internal state value of a RNG or without requiring its internal state information.

Disadvantages of using RNG refer to as True Random Number Generation

TRNGs are generally rather *inefficient* compared to PRNGs, taking considerably longer time to produce numbers. They are also *nondeterministic*, meaning that a given sequence of numbers cannot be reproduced, although the same sequence may, of course, occur several times by chance. TRNGs have no period, slower, more expensive and hardware-dependent solutions compared to PRNGs.

Comparison of TRNG and PRNG

The structural comparison of PRNG and TRNG number generators is shown in Table below. According to Table[5] below, PRNGs generate fast, easily designable, and periodic numbers. On the other hand, TRNGs generate unpredictable, entropy dependent, and nonperiodic numbers. Besides these advantages, they are disadvantageous compared to PRNGs because they are hardware dependent and operate slowly

[4] *https://www.hindawi.com/journals/cmmm/2018/3579275/*

[5] *www.random.org/randomness*

Comparison of TRNG and PRNG.		
	TRNG	PRNG
Realization type	Hardware required	Optional
Periodicity	Aperiodic	Periodic
Ease of application of design cycle	Complex	Easy because of standard structures
Efficiency	Weak	Perfect
Change of theoretical calculation limit	Constant (independent of time)	Dependent on time
Cryptographic security requirement	R1, R2 requirement	R1, R2, R3, and R4 requirements

Exercise 4

Implement a naive approach for generating random numbers in the set 0, 1, ..., 127. For this naive approach, generate a random 8-bit value, interpret that value as an integer, and reduce that value modulo 128. Experimentally generate 512 random numbers in the set 0, 1, ..., 127, and report on the distribution of results.

Answers:

In this question, we use the random.randint() to generate an array of 8 bits value (of 0 or 1). We then convert the binary values to an integer value. We then use the modulo 128 operation to derive a value which lies between 0 and 127. We use matplotlib package to plot the histogram for the distribution of 512 random values. In addition, we also generate a set of 5120, 51200, 512000 random values and their histograms. This is to illustrate that a larger sample size will "smoothen" the probability of each randomly generated value.

The codes are written in Python 2.7 environment.

^^

```
import matplotlib.pyplot as plt
import numpy as np
import random
def native():
    bit_array = np.random.randint(0, 2, size=8)
    print "The random 8 bit value is:", bit_array
    total = 0
    for i in range(8):
        total += bit_array[i] * 2**(8-1-i)
    res = total % 128
    print "The integer value is:", total
    print "The Mod 128 is:", res
    return res, total
native()
```

^^^

We ran the code and obtained a result as follows:

```
The random 8 bit value is: [1 1 0 1 1 0 1 1]
```

```
The integer value is: 219
The Mod 128 is: 91
```

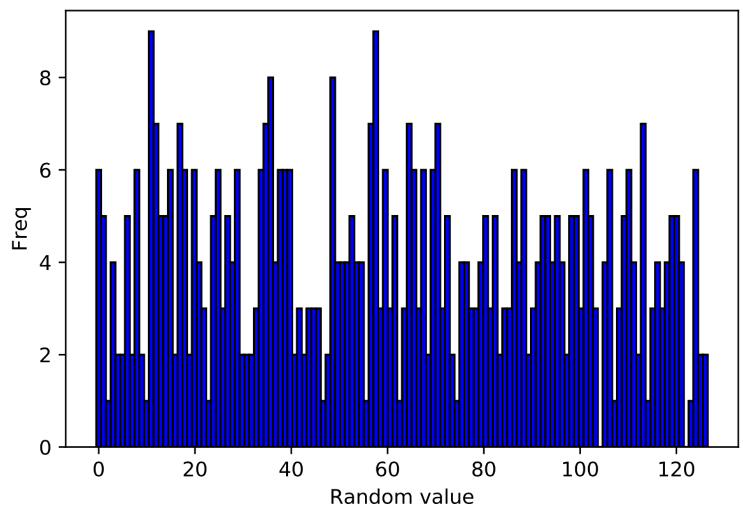
In the below codes, we continue to use the np.random.randint() to generate an array of 512 random values. We then plot the frequency histogram using the matplotlib library. We notice the uneven distribution of the random values showing unequal probabilities of the randomly generated numbers. This is due to the modulo bias problem. A 8 bit number generator would provide 256 values whose mod(128) would produce uneven distribution of random numbers between 0 to 127. There are several ways to avoid modulo bias and they can be found in stackoverflow - https://stackoverflow.com/questions/10984974/why-do-people-say-there-is-modulo-bias-when-using-a-random-number-generator . It is beyond the scope of this question and therefore we will not pursue a solution to resolve this. However, as a matter of interest, we increased the size of the numbers by factors of 10 (i.e. 5120), 100 (i.e. 51200) and 1000 (i.e. 512000) while maintaining the size to be divisible by the range size of 128. The histograms are included in the below results. The distribution of the random values is more even with higher numbers.

^^

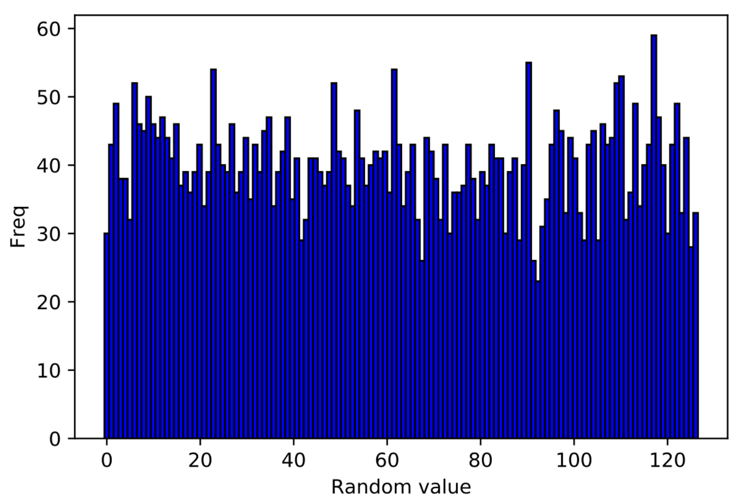
```
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
import numpy as np
import random
def native():
    bit_array = np.random.randint(2, size=8)
    total = 0
    for i in range(8):
        total += bit_array[i] * 2**(8-1-i)
    res = total % 128
    return res, total
def Multiple(num):
    number = []
    for i in range(num):
        number.append(native()[0])
    return number
```

```
histogram = Multiple(512)
figure(num=None, figsize=(10, 8), dpi=80, facecolor='w', edgecolor='k')
plt.hist(histogram, 128, color='blue', edgecolor='black', align='left')
plt.title('Histogram of Random Values')
plt.xlabel('Random Value')
plt.ylabel('Frequency')
```

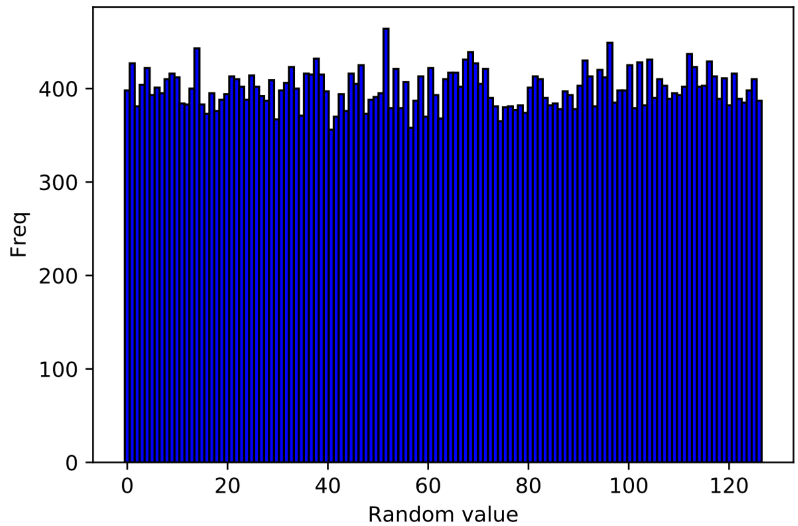
^^^



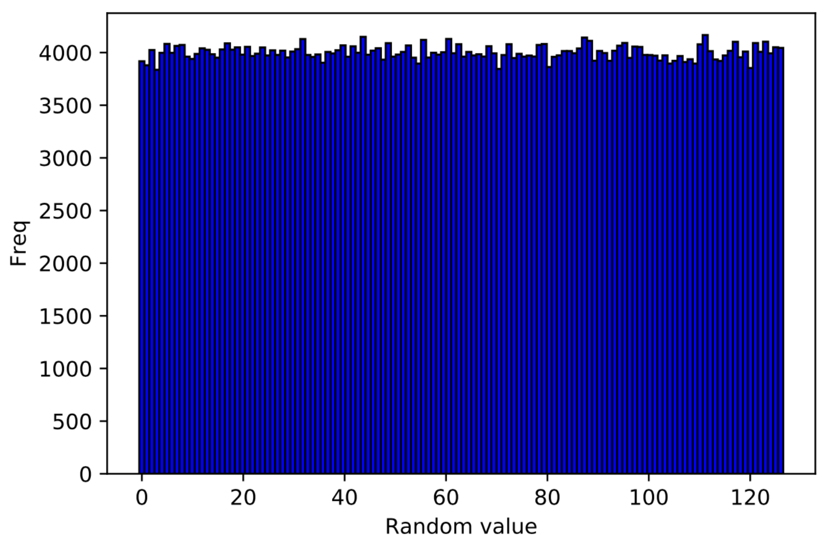
Sample = 512



Sample = 5120



Sample = 51200



Sample = 512000