HMAC CODE DOCUMENTATION

A Class named **HMAC** is created incorporating all required variables and functions.

```
class HMAC:
    def __init__(self,message,length):
        self.length = length
        self.func_key = ""
        self.message  = message
        self.opad = self.generate_message_identifier(24)
        self.ipad = self.generate_message_identifier(24)
        self.generate_key()
        self.hash_func = ""
```

Here length is the length of the fixed hash function upon which merkle Damgard transformation acts upon.

func_key is the **key_k** denoted by **k**
Opad and **ipad** are fixed distinct constants with respect to length n, here we have kept n' as 24 whereas n is equal to 16, here n is denoted by the variable length.

```
def generate_key(self):
    for i in range(24):
        self.func_key+=str(random.randint(0,1))
    merkle_damgard_hash = merkle_damgard(16)
    self.hash_func = merkle_damgard_hash
    self.hash_func.generate_key()
    return (self.func_key,self.hash_func.key_s)
```

This function is used to generate key **k** and key **s,** key s of the hash function as well as key k of length n' is generated and a tuple consisting both of them is returned.

```
def generate_message_identifier(self,length):
    message_identifier = ""
    for i in range(length):
        message_identifier+=str(random.randint(0,1))
    return message_identifier
```

This function is used to generate the message identifier (i.e opac and ipac) of the given length.

```
def encrypt(self,key_set,message):
    hash_key = key_set[1]
    mac_k = key_set[0]
    # print("mack = ",mac_k)
    self.hash_func.set_key(hash_key)
    str1 = bin((int(mac_k,2)^int(self.opad,2)))[2:]
    str2 = bin((int(mac_k,2))^int(self.ipad,2))[2:]+message
    self.hash_func.fill_value(str2,16)
    str3 = self.hash_func.chain_prop(hash_key)
    self.hash_func.fill_value(str1+str3,16)
    tag_t = self.hash_func.chain_prop(hash_key)
    # print("str1 = {}. str2 = {} str3 = {}".format(str1,str2,st
    return tag_t
    # tag_t = self.hash.fill_value()
```

This function takes input key **s** and key **k** and it encrypts the message using the following relation
Relevant outputs are stored in variable **str1**,**str2**,**str3**.

Mac: on input a key (s, k) and a message $m \in \{0,1\}^*$, output

$$t := H^s\Big((k \oplus \mathsf{opad}) \,\|\, H^s\big((k \oplus \mathsf{ipad}) \,\|\, m\big)\Big) .$$

The generated tag is stored in **tag_t** variable and is returned from this function.

```
def verify(self,key_set,tag,message):
    hash_key = key_set[1]
    mac_k = key_set[0]
    # print("mack = ",mac_k)
    self.hash_func.set_key(hash_key)
    str1 = bin((int(mac_k,2)^int(self.opad,2)))[2:]
    str2 = bin((int(mac_k,2))^int(self.ipad,2))[2:]+message
    self.hash_func.fill_value(str2,16)
    str3 = self.hash_func.chain_prop(hash_key)
    self.hash_func.fill_value(str1+str3,16)
    tag_t = self.hash_func.chain_prop(hash_key)
    # print("tag_t = {}, tag = {}".format(len(tag_t),len(tag)))
    if tag_t==tag:
        return 1
    else:
        return 0
```

Here key **s** key **k** , tag, and message are given as input to the function and this function follows the similar procedure as the above function except for the fact that here the final tag generated is compared to the initial given tag, if they are equal then our message and tag are verified else it is not verified.

```
if __name__=="__main__":
    # message = int(generate_key(17),2)
    message = generate_key(55)
    hmac = HMAC("",16)
    key_set = hmac.generate_key()
    tag = hmac.encrypt(key_set,message)
    print("generated key set(k,s) = ",key_set)
    print("generated tag = ",tag)
    # tag2= hmac.encrypt(key_set,message)
    # print("tag1 = {} , tag2 = {}".format(tag,tag2))
    # print("tag = ",tag)
    print("verifying message")
    print(hmac.verify(key_set,tag,message))
```

This part is used to give all the inputs and to call all the outputs the input bit, output bits are being printed on the screen,here the class is generated as well as keys and tags are generated and are verified.