VIETNAM NATIONAL UNIVERSITY-HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

FINAL PROJECT REPORT

SHA224-HMAC-HKDF

Student : **20C14001 – Le Duong Tuan Anh**

Teacher : **Assoc. Prof. Tran Minh Triet**

Dr. Truong Toan Thinh

Class : **Master of Information System**

Subject : **Cryptography**

Ho Chi Minh city, July 2021

Contents

1 General Information

Student Information

Student ID	Full Name	Email
20C14001	Le Duong Tuan Anh	leduongtuananh97@gmail.com

Report Information

This report is a summary of the **Final Project** based on **SHA224-HMAC-HKDF.**

2 SHA-224

SHA-224 is one of secure hash algorithms under standard Secure Hash Standard. This standard is first defined in document FIPS PUB 180-2. Any message with length less than 2^{64} bits can be an input to SHA-224. This hash algorithm provides output, called as a message digest, with length of 224 bits. This algorithm is an iterative one-way hash function.

Under FIPS PUB 180-4, SHA-224 is secure because, for a given algorithm, it is **computationally infeasible**:

- to find a message that corresponds to a given message digest.
- to find two different messages that produce the same message digest.

If the message is changed, with a very high probability, resulting in a different message digest. This will result in a verification failure when the secure hash algorithm is used with a digital signature algorithm or a keyed-hash message authentication algorithm.

SHA-224 is done within two stages: pre-processing *(padding, parsing and setting initialization values)* and hash computation *(generating a message schedule and series of hash vales)*. The final hash value is used to determine message digest.

2.1 Properties

SHA-224 has some following basic properties:

- Message Size: < 2^{64} bits.
- Block Size: 512 bits.
- Word Size: 32 bits.
- Message Digest Size: 224 bits.

2.2 Algorithm

This section covers algorithm of SHA-224, including needed functions, constants and so on. This project is simulated in Python 3 environment, so there will be coding example together.

2.2.1. Functions

Some basic functions, such as bitwise logical operation (AND/OR/XOR/NOT), modulo and right shift are pre-defined in Python 3. Following functions are needed for SHA-224 are implemented:

(1) *Rotate Right (circular right shift) operation*: **$ROTR^n(x)$**

x is a 32-bit word

n is an integer with 0 <= n < 32.

$ROTR^n(x) = (x << 32-n) OR (x >> n)$

```
# python3
wsize = 32
def ROTR(x, n):
    return ((x << (wsize - n)) & (2 ** wsize - 1)) | (x >> n)
```

In the implemented function, there is a change **in the first shift-left part**. Because the number definition in **Python is very large and shift-left will make number out of 32-bit range.** The **AND** operation let the **number back to 32-bit range**, keep the function also provide 32-bit value also.

(2) Some others needed functions, specific for SHA-224

All input parameters and output are 32-bits value

Formula	Python 3 Implementations
CH(x,y,z): (x AND y) XOR ((NOT x) AND Z)	`def CH(x,y,z):` `  return (x & y) ^ (~x & z)`
MAJ(x,y,z): (x AND y) XOR (x AND Z) XOR (y AND z)	`def MAJ(x,y,z):` `  return (x & y) ^ (x & z) ^ (y & z)`
BSIG0(x): $ROTR^{2}(x)$ XOR $ROTR^{13}(x)$ XOR $ROTR^{22}(x)$	`def BSIG0(x):` `  return ROTR(x,2) ^ ROTR(x,13) ^ ROTR(x,22)`
BSIG1(x): $ROTR^{6}(x)$ XOR $ROTR^{11}(x)$ XOR $ROTR^{25}(x)$	`def BSIG1(x):` `  return ROTR(x,6) ^ ROTR(x,11) ^ ROTR(x,25)`
SSIG0(x): $ROTR^{7}(x)$ XOR $ROTR^{18}(x)$ XOR (x SHR 3)	`def SSIG0(x):` `  return ROTR(x,7) ^ ROTR(x,18) ^ (x >> 3)`
SSIG1(x): $ROTR^{17}(x)$ XOR $ROTR^{19}(x)$ XOR (x SHR 10)	`def SSIG1(x):` `  return ROTR(x,17) ^ ROTR(x,19) ^ (x >> 10)`

2.2.2.Constants

(1) Round Constants (K)

There are 64 32-bit contanst values are used in SHA-224. Each value (0-63) is **the 32 bits of the fractional parts of the cube roots of the first 64 primes** (from 2 to 311). In my implementation, K is defined at below:

```
K = [
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
]
```

```
K[0]
```

```
1116352408
```

```
K[49]
```

```
506948616
```

Note: When coding I define in hex-style. Python automatically convert to decimal number when showing.

(2) Initialization H value

There are 8 32-bit contants also for initializated hash value. **Each value represents the first 32 bits of the fractional parts of the square roots of the first 8 primes**: **2, 3, 5, 7, 11, 13, 17, 19**.

```
H = [
    0xc1059ed8,
    0x367cd507,
    0x3070dd17,
    0xf70e5939,
    0xffc00b31,
    0x68581511,
    0x64f98fa7,
    0xbefa4fa4
]
```

```
K[0]
```

```
1116352408
```

```
K[4]
```

```
961987163
```

2.2.3. Preprocessing

We take message "HCMUS@2021" for example when working with SHA-224. The converted binary representation as below:

01001000 01000011 01001101 01010101 01010011 01000000 00110010 00110000 00110010 00110001

Step 1: Padding "1" value to original message

01001000 01000011 01001101 01010101 01010011 01000000 00110010 00110000 00110010 00110001 **1**

Step 2: Padding K "0"s to the message, *where K is the smallest, non-negative solution to the equation*

Length of Message + 1 + K = 448 (mod 512)

Now data is a multiple of 512, less 64 bits.

01001000 01000011 01001101 01010101 01010011 01000000 00110010 00110000 00110010 00110001 1**0000000 00000000**

Step 3: Append 64 bits to the end, *where the 64 bits are a big-endian integer representing the length of the original input in binary*. The length of original message is 80, which is represented in 64-bit binary as:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 **01010000**

Append this block to message block, we have length of the message will be a multiple of 512 bits.

01001000 01000011 01001101 01010101 01010011 01000000 00110010 00110000
00110010 00110001 10000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 01010000

Now we've done on preprocessing message. This message is processed through hash computation step.

2.2.4. Hash Computation

The length of message now is always devisible by 512. Firstly, we need to initialize the Hash Values (H) as the constant above *(since this step, we will describe more clearly in hex-representation for simplification)*.

```
H = [    0xc1059ed8, 0x367cd507, 0x3070dd17, 0xf70e5939,
         0xffc00b31, 0x68581511, 0x64f98fa7, 0xbefa4fa4 ]
```

For all following steps, **SHA-224 performs on every 512-bit "chunk" of the message**. **For every iteration, the computation updates the Hash Value H.** *In this example, the message is quite short and can be processed for only 1 iteration.*

Step 1: Create Message Schedule

The **Message Schedule** just does the split for chunk data to every 32-bits value. This is required as all of our functions are calculated on 32-bits. **Message Schedule** is stored as array with 64 elements: w[0..63]

For the first 16 values, it is copied from chunk data:

```
for t in range(16):
    w[t] = chunk[t*wsize:(t+1)*wsize]
```

01001000010000110100110101010101 01010011010000000011001000110000
00110010001100011000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000001010000000

For the next 48 values, it is calculated as following formula:

```
for t in range(16, 64):
    w[t] = SSIG1(w[t-2]) + w[t-7] + SSIG0(w[t-15]) + w[t-16])
    w[t] = w[t] % 2**wsize
```

The second line is just used for truncating number to 32-bit value.

For example, t = 16

```
SSIG1(w[14])  = 00000000000000000000000000000000
w[9]          = 00000000000000000000000000000000
SSIG0(w[1])   = 01100110010000101001001011110010
w[0]          = 01001000010000110100110101010101
Sum up all:
w[16]         = 10101110100001011110000001000111
```

Now we will have following result, including 64 values in our **message schedule**:

01001000010000110100110101010101 01010011010000000011001000110000

00110010001100011000000000000000 00000000000000000000000000000000

00000000000000000000000000000000 00000000000000000000000000000000

00000000000000000000000000000000 00000000000000000000000000000000

00000000000000000000000000000000 00000000000000000000000000000000

00000000000000000000000000000000 00000000000000000000000000000000

00000000000000000000000000000000 00000000000000000000000000000000

00000000000000000000000000000000 00000000000000000000001010000

10101110100001011110000001000111 10111001100001001001000110111100

01111110001100100000001111101010 11011010110001111010101011010110

01000001100101111111110001011111 00100000000001110000011111010001

00000001101101000010110101010110 00010001101000001001011000001001

...

00000100011110111010000000100100 11011000110011101100000011100111

// 64 values total

Step 2: Initialize the working variables, from H: a = H[0], b = H[1], c = H[2]...h=H[7]

```
a,b,c,d,e,f,g,h = H // In Python, we just need to spread out
```

Step 3: Perform the main hash computation, **in 64 times.**

```
for t in range(64):
    T1 = (h + BSIG1(e) + CH(e,f,g) + K[t] + w[t]) % 2**wsize
    T2 = (BSIG0(a) + MAJ(a,b,c)) % 2**wsize
    h = g
    g = f
    f = e
    e = (d + T1) % 2**wsize
    d = c
    c = b
    b = a
    a = (T1 + T2) % 2**wsize
```

Note: The modulo is used to keep result in 32-bit range.

For example, in the 1st iteration:

```
// Extract from H
a = 0xc1059ed8
b = 0x367cd507
c = 0x3070dd17
d = 0xf70e5939
e = 0xffc00b31
f = 0x68581511
g = 0x64f98fa7
h = 0xbefa4fa4
```

```
# Calculate value inside the loop
T1 = 0xf406b2fa
T2 = 0x0170e9b5
a = 0xf5779caf
b = 0xc1059ed8
c = 0x367cd507
d = 0x3070dd17
e = 0xeb150c33
f = 0xffc00b31
g = 0x68581511
h = 0x64f98fa7
```

Step 4: Update the H array, from intermediate hash value:

```
H[0] = (H[0] + a) % 2**wsize
H[1] = (H[1] + b) % 2**wsize
H[2] = (H[2] + c) % 2**wsize
H[3] = (H[3] + d) % 2**wsize
H[4] = (H[4] + e) % 2**wsize
H[5] = (H[5] + f) % 2**wsize
H[6] = (H[6] + g) % 2**wsize
H[7] = (H[7] + h) % 2**wsize
```

After all computations are done for all chunks in the input message. The output is produced from concatenation of H. In SHA-224, this is this is the concatenation of H[0..6] **(7 values x 4 bytes = 28 bytes in digest message.**

The hashed value for "HCMUS@2021" is:

66c91ad87db650f856005bdd6a5a31712eb94c7c7987cac2012ff012

2.3 Results

In this section, some comparisons are done between our implemented version, versus default hasing library of Python 3: **hashlib**. This library also supports SHA-224 and other hash algorithms also.

Case 1: "HCMUS@2021".

```
st = "HCMUS@2021"
sha_224(st.encode('utf-8')).hex()
```

```
'66c91ad87db650f856005bdd6a5a31712eb94c7c7987cac2012ff012'
```

```
import hashlib
hashlib.sha224(st.encode('utf-8')).digest().hex()
```

```
'66c91ad87db650f856005bdd6a5a31712eb94c7c7987cac2012ff012'
```

Case 2: Empty string ""

```
st = ""
sha_224(st.encode('utf-8')).hex()
```

```
'd14a028c2a3a2bc9476102bb288234c415a2b01f828ea62ac5b3e42f'
```

```
import hashlib
hashlib.sha224(st.encode('utf-8')).digest().hex()
```

```
'd14a028c2a3a2bc9476102bb288234c415a2b01f828ea62ac5b3e42f'
```

Case 3: Long string that needed to iterate through many chunks, in this case, 2 chunks:

"This is a very long message to evaluate SHA-224, done by HCMUS STUDENT!_@_#_$_%_^_&_*"

```
st = "This is a very long message to evaluate SHA-224, done by HCMUS STUDENT!_@_#_$_%_^_&_*"
sha_224(st.encode('utf-8')).hex()
```

```
'1422c3b4f3b9dc93831ff732011cf260eaff5ebf4d7d5bcfeb7f69ab'
```

```
import hashlib
hashlib.sha224(st.encode('utf-8')).digest().hex()
```

```
'1422c3b4f3b9dc93831ff732011cf260eaff5ebf4d7d5bcfeb7f69ab'
```

Case 4: Much longer string, hashing through 25 chunks:

"abcd1234@#$%..... abcd1234@#$%" *(repeated 128 times)*

```
st = "abcd1234@#$%"*128
sha_224(st.encode('utf-8')).hex()
```

```
'a3ebd32627426cb20637b527d4397ead69a450624a0bffabd752dab5'
```

```
import hashlib
hashlib.sha224(st.encode('utf-8')).digest().hex()
```

```
'a3ebd32627426cb20637b527d4397ead69a450624a0bffabd752dab5'
```

3 Keyed-hash Message Authentication Code

Key-Hash Message Authentication Code (HMAC) is a mechanism for message authentication, using cryptographic hash function. A hash function must be an iterative approved hash function, such as SHA-224, SHA-256, so on. In this report, HMAC is used with SHA-224 and a shared secret key.

HMAC provides a way to check the integrity of information, based on a shared secret key. This kind of mechanism are used call as Message Authentication Code (MAC). The "H" just describes the approved Hash function.

3.1 Properties

HMAC includes following things:

- *Hash function*: an approved cryptographic hash function, to map a string to a fixed length string.
- *Secret key:* a cryptographic key that is uniquely associated to some entities.
- *Message*: the message needs to be encrypted.

3.2 Algorithm

3.2.1.Parameters

Beside Secret Key and input message, HMAC uses following parameters inside the function:

- **B:** Block Size, size of input to the approved hash function. In this report, it is 64 bytes, SHA-224.
- **H:** Hash function - SHA-224
- **K_0:** The key, generated from the secret key (K). This key is exact B-byte.
- **ipad**: The inner pad, the byte *0x36*, repeated B times.
- **opad**: The outer pad, the byte *0x5c*, repeated B times.

The *ipad* is 0x36 = **0b0110110**

The *opad* is 0x5c = **0b1011100**

In the HMAC paper (https://cseweb.ucsd.edu/~mihir/papers/kmd5.pdf), high Hamming distance between these pads are preferred. The hamming distance between *ipad* and *opad* are 4, not too high (max = 8). I think the reason is:

- If distance is low, *ipad* ~ *opad*, key to H is similar → not good.
- If distance is high, *ipad* ~ (**NOT** *pad*), there is also a relationship and leads to above reason.

That's why the values of *ipad/opad* were chosen.

3.2.2.Algorithm

The formula of HMAC is described as below:

HMAC(K, text) = **H**((**K_0** XOR *opad*) + **H**((**K_0** XOR *ipad*) + *text*))

In this project, we implement **HMAC** by following step:

Step 1: Find K_0

This HMAC is based on SHA-224, so Block Size = 64. If the length of K larger than 64, it must be hashed first, by SHA-224. Then padding zero's to the K is needed if length is less than 64.

```
B = 64
# hash if K is longer than B
if len(key) > B:
    key = sha_224(key)
# pad zero's if K is less than B
key = key + b'\x00'*(B - len(key))
```

Step 2: Generate and apply XOR operation for *ipad/opad*. The *ipad/opad* is first initialized and XOR with the K_0.

```
ipad = bytes((x ^ 0x36) for x in key)
opad = bytes((x ^ 0x5c) for x in key)
```

Step 3: Hash the concatenated value between *ipad* and *text*, using SHA-224.

```
ipad_msg = sha_224(ipad + msg)
```

Step 4: Generated the **HMAC** value, by hasing concatenated value between *opad* and *ipad_msg*, using SHA-224.

```
hmac = sha_224(opad + ipad_msg)
```

For example, key = "hcmus", message = "abcd1234"

B = 64

K = 0x68636d7573

It is not enough, so padding zeros, assigned to K_0

K_0 = 0x68636d757300000000000…0000 // size=B

ipad = 0x36363636363636363636…363636 // size=B

opad = 0x5c5c5c…5c5c5c5c5c5c5c5c5c5c5c // size=B

Now XOR the ipad/opad with K_0.

ipad = 0x5e555b4345363636363636363636363636…3636

opad = 0x343f31292f5c5c5c5c5c5c5c5c5c5c5c5c5c…5c5c

The message is:

msg = 0x6162636431323334

The concatenation (red is *ipad*, the blue is from message)

ipad + msg = 0x**5e555b43453636363636…3636366162636431323334**

Hash above number by SHA-224, we get **28-byte value** as below:

ipad_msg =
0x8ba7cac166586688c698ca6ec9b4beaa3d27ac1fdc1f5133807d0a66

Concatenate the *opad* with above hashed value, we get following string (red is *opad*, the blue is hashed value):

0x**343f31292f5c5c5…5c5c5c8ba7cac166586688c698ca6ec9b4beaa3d27ac1fdc1f5133807d0a66**

Hash this string by SHA-224, we get the HMAC value:

2610f96b7e7baf6a841d03c2b88fa79b003754dad906b17d0a16b866

3.3 Results

In this section, some comparisons are done between our implemented version, versus default **hmac** library of Python 3. This library supports **hmac**, with the specific hashing function. We pass the SHA-224 from **hashlib** to verify the result.

Case 1: key = "hcmus", message = "abcd1234".

```
key = 'hcmus'
st = 'abcd1234'
hmac_sha224(key=key.encode('utf-8'),msg=st.encode('utf-8')).hex()
```

```
'2610f96b7e7baf6a841d03c2b88fa79b003754dad906b17d0a16b866'
```

```
import hmac
hmac.new(key=key.encode('utf-8'), msg=st.encode('utf-8'), digestmod=hashlib.sha224).digest().hex()
```

```
'2610f96b7e7baf6a841d03c2b88fa79b003754dad906b17d0a16b866'
```

Case 2: Empty both key/message

```
key = ''
st = ''
hmac_sha224(key=key.encode('utf-8'),msg=st.encode('utf-8')).hex()
```

```
'5ce14f72894662213e2748d2a6ba234b74263910cedde2f5a9271524'
```

```
import hmac
hmac.new(key=key.encode('utf-8'), msg=st.encode('utf-8'), digestmod=hashlib.sha224).digest().hex()
```

```
'5ce14f72894662213e2748d2a6ba234b74263910cedde2f5a9271524'
```

Case 3: Key size=64. key = "abcd1234@#$%()*&... abcd1234@#$%()*&", repeated 4 times. Message = "HCMUS@2021".

```
key = 'abcd1234@#$%()*&'*4
st = 'HCMUS@2021'
hmac_sha224(key=key.encode('utf-8'), msg=st.encode('utf-8')).hex()
```

```
'fc916a37d69f804c961f409f6979800d7625167cd42e358db00e6cac'
```

```
import hmac
hmac.new(key=key.encode('utf-8'), msg=st.encode('utf-8'), digestmod=hashlib.sha224).digest().hex()
```

```
'fc916a37d69f804c961f409f6979800d7625167cd42e358db00e6cac'
```

Case 4: Key is longer than B. key = "abcd1234@#$%()*&... abcd1234@#$%()*&", repeated 16 times. Message = "HCMUS@2021... HCMUS@2021" repeated 512 times

```
key = 'abcd1234@#$%()*&' * 16
st = 'HCMUS@2021' * 512
hmac_sha224(key=key.encode('utf-8'), msg=st.encode('utf-8')).hex()
```

```
'105976e7a9d72d1578ece881335671441cf84dcff393937aafd0da63'
```

```
import hmac
hmac.new(key=key.encode('utf-8'), msg=st.encode('utf-8'), digestmod=hashlib.sha224).digest().hex()
```

```
'105976e7a9d72d1578ece881335671441cf84dcff393937aafd0da63'
```

4 HMAC-based Extract-and-Expand Key Derivation Function

HMAC-based *Extract-and-Expand* Key Derivation Function (HKDF) is a famous mechanism in building block in various protocols and application. The **Key Derivation Function** is used to take some source of initial keying material, derive from it one or more cryptographically strong secret keys.

HKDF follows the "extract-and-expands" paradigm, consists of 2 modules for each stage:

- *Stage 1 – Extractions:* This module takes the input, including keying material, **extracts** from it to **a fixed-length pseudorandom** key K. This utilizes **the diffusion** properties of HMAC.
- *Stage 2 – Expansion:* This module **expands** the key K **into several additional pseudorandom keys**. Depends on defined length, these keys are used to generate the output of the **KDF**. The multiple keys are produced deterministically from the initial shared key, so that **the same process may produce those same secret keys safely on multiple devices**, from the same input.

4.1 Properties

Each stage in HKDF needs different parameters.

Stage 1: Extract

- *Hash function*: HMAC function. In this project, we use HMAC, based on SHA-224.
- *Salt*: Salt value (a non-secret random value). If not set, this is a string of zero.
- *IKM*: Input keying material

Stage 2: Expansion

- *Hash function*: HMAC function. In this project, we use HMAC, based on SHA-224.
- *PRK*: a pseudorandom key. **This is usually from step 1**.
- *info*: Optional context and application specific information.
- *L*: length of output keying material.

4.2 Algorithm

4.2.1.Parameters

HKDF uses following parameters for each step:

Stage 1: Extract

- **H:** Hash length of hashed value from HMAC. In this report, it is 28 bytes, SHA-224.

Stage 2: Expansion

- **H:** Hash length of hashed value from HMAC. In this report, it is 28 bytes, SHA-224.
- **n:** n = ceil(L / H), determines the maximum chunks for all OKM.

4.2.2.Algorithm

Stage 1: Extract

```
hash_length = 28 # Hashlength SHA-224
// If the salt value is not defined, generate the 0's salt
if len(salt) == 0:
     salt = bytes(0x0 * hash_length)
prk = hmac_sha224(
     key=salt,
     msg=ikm,
)
```

Stage 2: Expansion

```
hash_length = 28 # Hashlength SHA-224
// The length of output keying material must less than 255*H
if (L > 255*hash_length):
     raise Exception('Error: L <= 255*hash_length')
n = ceil(L/hash_length)
// T = T(1) + T(2) + … + T(n)
t = b""
okm = b""
// T[i] = HMAC(PRK, T[i-1] + info + 0xi
for i in range(n):
     t = hmac_sha224(key=prk, msg=(t + info + bytes([1+i])))
     okm += t
// Actual result is the first L value of concatenated output.
Result = okm[:L]
```

4.3 Result

There is no default implemented HKDF in Python, so we will compare our implemented version, versus an online version, https://asecuritysite.com/encryption/HKDF.

Case 1: Empty of all inputs. Key length L = 16.

Tool's result:

```
Hashing type:   SHA-224
Message:
  Hex:
Salt:
Info:
==================
PRK: 5ce14f72894662213e2748d2a6ba234b74263910cedde2f5a9271524
OKM: ba93ac4d2ed54868a9192c04ca065366
```

Our result:

```
1  ikm = ''.encode('utf-8')
2  salt = ''.encode('utf-8')
3  prk = hkdf_extract(ikm, salt)
4  print('prk =', prk.hex())
5
6  info = ''.encode('utf-8')
7  L = 16
8  okm = hkdf_expand(prk, L, info)
9  print('okm =', okm.hex())
```

```
prk = 5ce14f72894662213e2748d2a6ba234b74263910cedde2f5a9271524
okm = ba93ac4d2ed54868a9192c04ca065366
```

Case 2:

ikm = 'hello world'

salt = '0x0123456789ABCDEF'

info = '0x9876543210'

L = 16

Tool's result:

```
Hashing type:   SHA-224
Message:         hello world
  Hex:           68656c6c6f20776f726c64
Salt:            0123456789abcdef
Info:            9876543210
====================
PRK: 1692b4717241200686300 27fd60767fe312ea711c9e969f806e8780e
OKM: 820644d484ab8c00bbe2f4fd98cffdd2
```

Our result:

```python
ikm = 'hello world'.encode('utf-8')
salt = bytes.fromhex('0123456789abcdef')

prk = hkdf_extract(ikm, salt)
print('prk =', prk.hex())

info = bytes.fromhex('9876543210')
L = 16
okm = hkdf_expand(prk, L, info)
print('okm =', okm.hex())
```

```
prk = 1692b4717241200686300 27fd60767fe312ea711c9e969f806e8780e
okm = 820644d484ab8c00bbe2f4fd98cffdd2
```

Case 3:

ikm = 'HCMUS@2021' repeated 64 times

salt = '0xffffffff

info = '0x00000000

L = 16

Tool's result:

```
Hashing type:   SHA-224
Message:
HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMU
S@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@20
21HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HC
MUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@
2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021
HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMU
S@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021HCMUS@2021
  Hex:
Salt:           ffffffff
Info:           00000000
==================
PRK: 1d1bae81b9115d50e063bd56f7b47f7645a04c31e2434e0c59e8914a
OKM: e195cd5d8c7d5177d493d3d4ee93129e
```

Our result:

```python
ikm = ('HCMUS@2021'*64).encode('utf-8')
salt = bytes.fromhex('ffffffff')

prk = hkdf_extract(ikm, salt)
print('prk =', prk.hex())

info = bytes.fromhex('00000000')
L = 16
okm = hkdf_expand(prk, L, info)
print('okm =', okm.hex())
```

```
prk = 1d1bae81b9115d50e063bd56f7b47f7645a04c31e2434e0c59e8914a
okm = e195cd5d8c7d5177d493d3d4ee93129e
```

Case 4:

ikm = 'HCMUS@2021'

salt = '0xffffffffabcdef123456' repeated 128 times

info = '0x00000000

L = 16

Tool's result:

```
Hashing type:    SHA-224
Message:         HCMUS@2021
  Hex:           48434d55534032303231
Salt:            ffffffffabcdef123456...
Info:            00000000
==================
PRK: 6041c95f563ed2bee0911172c2362e958ec42516ae0e9dd6957f65a5
OKM: c5b83b5957c6ab112813d359dc81bfc0

Key (Hex):       c5b83b5957c6ab112813d359dc81bfc0
Key (Base-64):   xbg7WVfGqxEoE9NZ3IG/wA==
```

Our result:

```python
ikm = ('HCMUS@2021').encode('utf-8')
salt = bytes.fromhex('ffffffffabcdef123456'*128)

prk = hkdf_extract(ikm, salt)
print('prk =', prk.hex())

info = bytes.fromhex('00000000')
L = 16
okm = hkdf_expand(prk, L, info)
print('okm =', okm.hex())
```

```
prk = 6041c95f563ed2bee0911172c2362e958ec42516ae0e9dd6957f65a5
okm = c5b83b5957c6ab112813d359dc81bfc0
```

Case 5:

ikm = 'HCMUS@2021'

salt = '0xffffffff'

info = '0x0123456789fedcba'

L = 8

Tool's result:

```
Hashing type:    SHA-224
Message:         HCMUS@2021
  Hex:           48434d55534032303231
Salt:            ffffffff
Info:            0123456789fedcba
==================
PRK: 88c970a4f798684a1100e5fdd55ea3ec99181a51d4c6fb5a98fdd626
OKM: 1c30d7e32670e883

Key (Hex):       1c30d7e32670e883
Key (Base-64):   HDDX4yZw6IM=
```

Our result:

```python
ikm = ('HCMUS@2021').encode('utf-8')
salt = bytes.fromhex('ffffffff')

prk = hkdf_extract(ikm, salt)
print('prk =', prk.hex())

info = bytes.fromhex('0123456789fedcba')
L = 8
okm = hkdf_expand(prk, L, info)
print('okm =', okm.hex())
```

```
prk = 88c970a4f798684a1100e5fdd55ea3ec99181a51d4c6fb5a98fdd626
okm = 1c30d7e32670e883
```

Case 6:

ikm = 'HCMUS@2021'

salt = '0xffffffff'

info = '0x0123456789fedcba'

L = 16

Tool's result:

```
Hashing type:   SHA-224
Message:        HCMUS@2021
  Hex:          48434d55534032303231
Salt:           ffffffff
Info:           0123456789fedcba
==================
PRK: 88c970a4f798684a1100e5fdd55ea3ec99181a51d4c6fb5a98fdd626
OKM: 1c30d7e32670e883af4f76fee54313db

Key (Hex):      1c30d7e32670e883af4f76fee54313db
Key (Base-64):  HDDX4yZw6IOvT3b+5UMT2w==
```

Our result:

```python
ikm = ('HCMUS@2021').encode('utf-8')
salt = bytes.fromhex('ffffffff')

prk = hkdf_extract(ikm, salt)
print('prk =', prk.hex())

info = bytes.fromhex('0123456789fedcba')
L = 16
okm = hkdf_expand(prk, L, info)
print('okm =', okm.hex())
```

```
prk = 88c970a4f798684a1100e5fdd55ea3ec99181a51d4c6fb5a98fdd626
okm = 1c30d7e32670e883af4f76fee54313db
```

Case 7:

ikm = 'HCMUS@2021'

salt = '0xffffffff'

info = '0x0123456789fedcba'

L = 64

Tool's result:

```
Hashing type:    SHA-224
Message:          HCMUS@2021
  Hex:            48434d55534032303231
Salt:             ffffffff
Info:             0123456789fedcba
====================
PRK: 88c970a4f798684a1100e5fdd55ea3ec99181a51d4c6fb5a98fdd626
OKM:
1c30d7e32670e883af4f76fee54313dbf69abaf5834c55ad602b96cdadbde77128c7c3e2a2a65e175cd2daa9b09ab1
e877c889d42f04d87d5fb8274098b4a04d
```

Our result:

```
1  ikm = ('HCMUS@2021').encode('utf-8')
2  salt = bytes.fromhex('ffffffff')
3
4  prk = hkdf_extract(ikm, salt)
5  print('prk =', prk.hex())
6
7  info = bytes.fromhex('0123456789fedcba')
8  L = 64
9  okm = hkdf_expand(prk, L, info)
10 print('okm =', okm.hex())
```

```
prk = 88c970a4f798684a1100e5fdd55ea3ec99181a51d4c6fb5a98fdd626
okm = 1c30d7e32670e883af4f76fee54313dbf69abaf5834c55ad602b96cdadbde77128c7c3e2a2a65e175cd2daa9b09ab1e877c889d42f04d
87d5fb8274098b4a04d
```

5 Conclusion

In this report, we have implemented end-to-end Python version, for SHA-224, HMAC and HKDF. In this section, we give some comments about these functions.

- Comparing with earlier hashing type, **SHA-224 is more secure than MD5 and SHA-1**, **but in term of performance, it is slower**.
- **SHA-2 hashing functions is not much different from SHA-1**, so it might be cracked soon.
- **SHA-3 is recently announced**, and this is a good replacement for SHA-2 in the near future.
- HMAC uses a symmetric key**, which is both used for encryption and decryption**. So, the exchange is needed to consider.
- If the symmetric key of HMAC is shared between multiple parties**, there is a possibility** that **someone which has the key will make a fraud message**.