Problem Sheet 8

Problem 8.1

Solution:

a)
300974085 + 30002011 = 330976096

b)
In case if your private key got compromised or you lost access to it, and you do not want other people send emails that are encrypted with the corresponding public key. You can create revocation certificate in order to be able to ”disable” the key in case of the previously mentioned events. You would upload revoked key file to the server, and that would notify people trying to use your public key that it is revoked (also it might help to send your revoked key to frequent receivers). [http://www.spywarewarrior.com/uiuc/ss/revoke/pgp-revoke.htm]

Problem 8.2

Solution:

a)
Function that takes secret, seed and an identifying label and produces output of arbitrary length.

b)
In the key exchange in TLS 1.2 the premaster is the randomly generated data that is later used to compute Master secret. It is send by client using RSA encryption with public key of the server to the server. After that exchange both server and client compute the masterkey using

```
PRF(pre_master_secret, "master secret",ClientHello.random + ServerHello.random)[0..47];
```

After that premaster secret is discarded.
A conventional Diffie-Hellman computation is performed, where the negotiated key is premaster secret, and the computed key is the master secret.
After that the masterkey is used to generate Finished messgaes, encryption keys, and MAC keys.
The reason why both exist, lies within the unconsistency of the cipher suites and some expectations of the master secret.
Master secret is desired to be pseudorandom and also have almost no structure for security reasons. And that is provided immediately by RSA suite, but some other suites different methods to generate the secret and have different lengths which might be poorly distributed. If we have both master key and premaster, we would be able to have master key in the exact way we want consistently and all the difference between ciphers are only noticed on the level of premaster key. [https://crypto.stackexchange.com/questions/24780/what-is-the-purpose-of-pre-master-secret-in-ssl-tls]

c) We use PRF

```
PRF(SecurityParameters.master_secret, "key expansion",
 SecurityParameters+ SecurityParameters.client_random);
```

until we generate enough data. Then we cut data into 6 other keys (some might be empty depending on the cipher used):

```
client_write_MAC_key[SecurityParameters.mac_key_length]
server_write_MAC_key[SecurityParameters.mac_key_length]
client_write_key[SecurityParameters.enc_key_length]
server_write_key[SecurityParameters.enc_key_length]
client_write_IV[SecurityParameters.fixed_iv_length]
server_write_IV[SecurityParameters.fixed_iv_length]
```

One of three ciphers is used for encryption : stream cipher, block sipher or AEAD.
Mac write key is used in generation of MAC for stream cipher and block cipher.Block sipher takes IV too. AEAD uses write key and no mac key. .

Problem 8.3

Solution:

a)
The logic is first to take a secret and extract good pseudorandom key out of it, since as I mentioned before some keys have non uniform distribution. This step increases the cryptographical strength of the result key. Then we expand it to the desired lengths, and ”cut” our keys from the expanded output.
Extract:
Extract takes two arguments - salt and IKM, computes and return the HMAC-Hash(Hash is some hash function)of them. Salt is optional value (non-secret random value), and by default it is set to string of HashLen zeros. IKM is input keying material.
Expand:
HKDF-Expand(PRK, info, L) -¿ OKM
PRK - pseudorandom key as an argument
info - optional context
L - length of output.
OKM - output keying material.
It is calculated using following formulas:

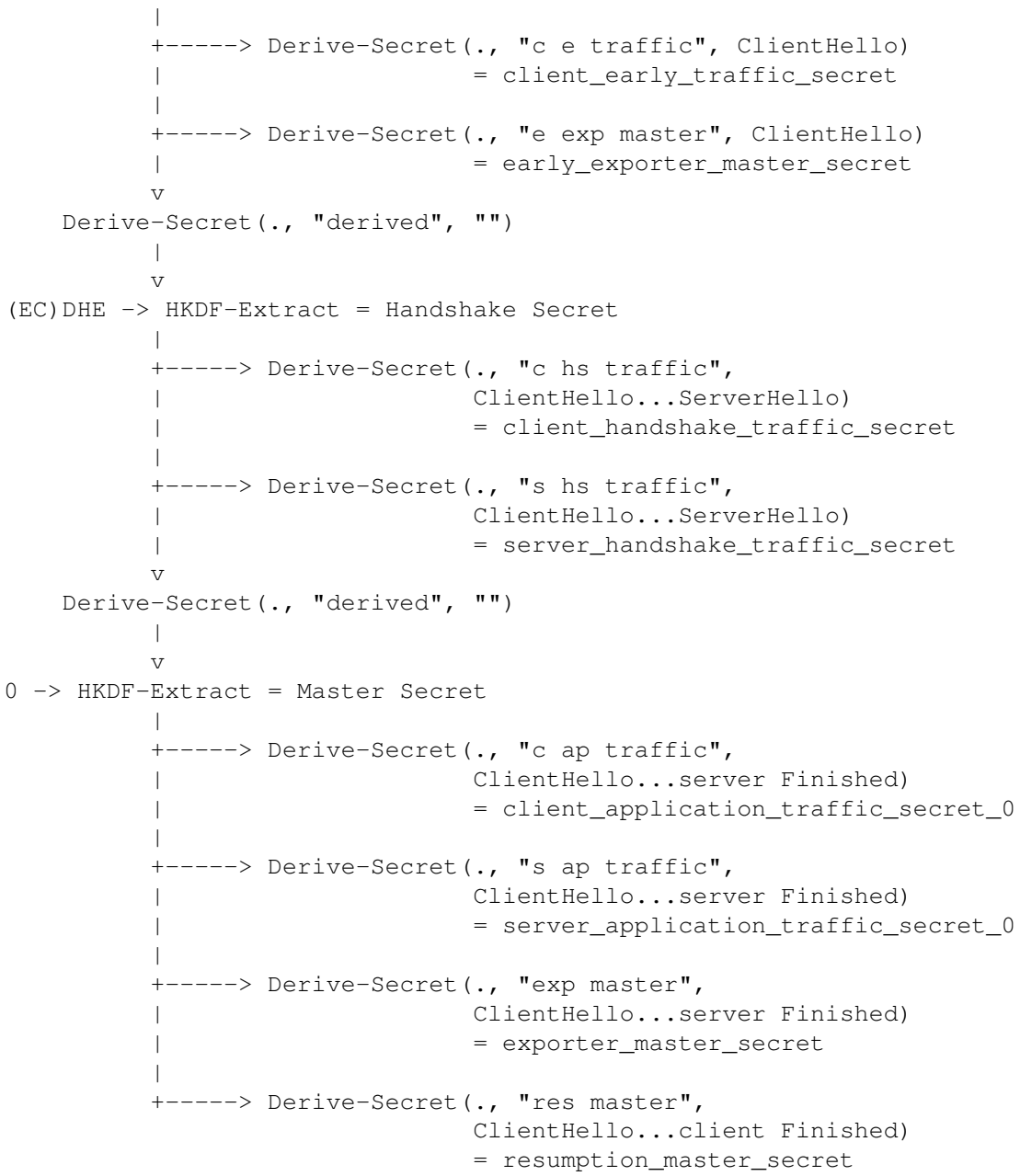
```
N = ceil(L/HashLen)
T = T(1) | T(2) | T(3) | ... | T(N)
OKM = first L octets of T

where:
T(0) = empty string (zero length)
T(1) = HMAC-Hash(PRK, T(0) | info | 0x01)
T(2) = HMAC-Hash(PRK, T(1) | info | 0x02)
T(3) = HMAC-Hash(PRK, T(2) | info | 0x03)
...
```

b)
RFC reference
TLS 1.3 takes advantage of HKDF to extract keys from master secret.
First it uses HKDF-Extract with salt being current secret state and IKM being a new secret.
2 keys are needed to be added - pre shared key and (EC)DHE shared secret.
Then we apply Derive-Secret defined as expand call with changed input parameters to derive different keys or salt for the next step.
Derive-secret ralted definitons:

```
HKDF-Expand-Label(Secret, Label, Context, Length) =
       HKDF-Expand(Secret, HkdfLabel, Length)

  Where HkdfLabel is specified as:

  struct {
      uint16 length = Length;
      opaque label<7..255> = "tls13 " + Label;
      opaque context<0..255> = Context;
  } HkdfLabel;

  Derive-Secret(Secret, Label, Messages) =
       HKDF-Expand-Label(Secret, Label,
                         Transcript-Hash(Messages), Hash.length)
```

And the flow is following:

```
            0
            |
            v
PSK ->  HKDF-Extract = Early Secret
            |
            +-----> Derive-Secret(., "ext binder" | "res binder", "")
            |                     = binder_key
```


|
+-----> Derive-Secret(., "c e traffic", ClientHello)
| = client_early_traffic_secret
|
+-----> Derive-Secret(., "e exp master", ClientHello)
| = early_exporter_master_secret
v
Derive-Secret(., "derived", "")
|
v
(EC)DHE -> HKDF-Extract = Handshake Secret
|
+-----> Derive-Secret(., "c hs traffic",
| ClientHello...ServerHello)
| = client_handshake_traffic_secret
|
+-----> Derive-Secret(., "s hs traffic",
| ClientHello...ServerHello)
| = server_handshake_traffic_secret
v
Derive-Secret(., "derived", "")
|
v
0 -> HKDF-Extract = Master Secret
|
+-----> Derive-Secret(., "c ap traffic",
| ClientHello...server Finished)
| = client_application_traffic_secret_0
|
+-----> Derive-Secret(., "s ap traffic",
| ClientHello...server Finished)
| = server_application_traffic_secret_0
|
+-----> Derive-Secret(., "exp master",
| ClientHello...server Finished)
| = exporter_master_secret
|
+-----> Derive-Secret(., "res master",
ClientHello...client Finished)
= resumption_master_secret