TLS1.3 设计过程

说明：

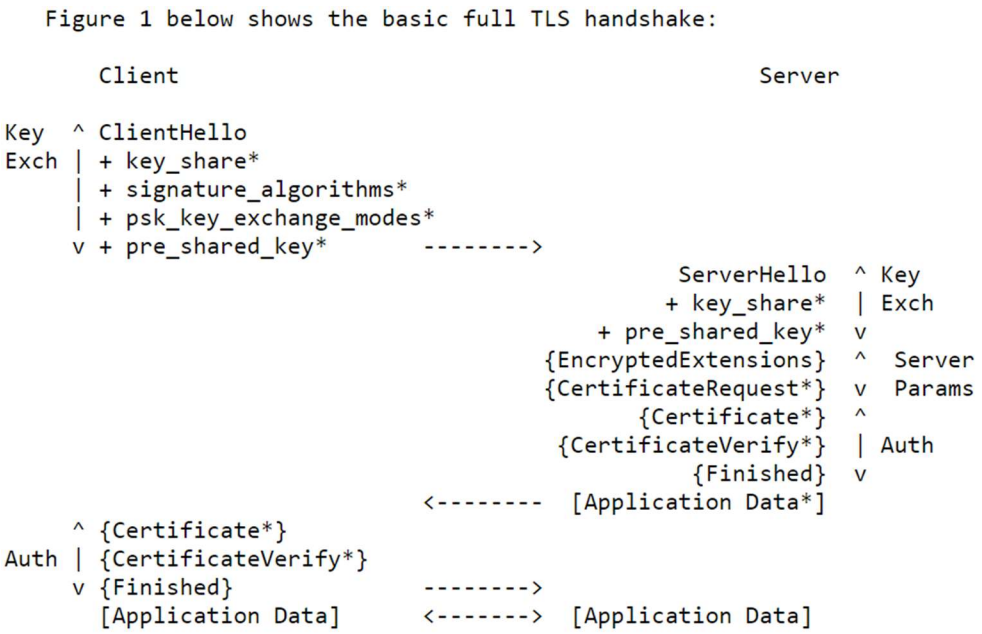
所有的数据结构以及字符表示参考都基于 **RFC8446**（**https://tools.ietf.org/html/rfc8446**）**,**为了方便阅读，以及以后的进一步完善，所有代码的命名格式也与 **RFC8446** 文档中伪代码以及说明名称一致，所以名称都较长。因此，代码看起来可能有点乱，辛苦助教了。

设计目标:

尽可能完成一个能够直接线上部署的简单的 **TLS1.3** 加密协议，希望与现实中在线的系统能够交互。使用者可以只需修改数据接口，便可以直接部署在真实网络中实现 **TLS1.3** 协议。

设计大纲:

此 **PJ** 中 **TLS1.3** 实现过程，与给定实验说明并不一样，实验指导中给的是完整的 **TLS1.3 1RTT** 的实现过程，但是现在大多数通信双方一个是服务器一个是客户端，一般服务器是线上产品有 **CA** 机构签发的完整证书，而客户端一般为个人用户，没有完整的 **CA** 证书，所以我根据协议内容，并没有完成服务器请求客户端提供证书。所以我实现的过程如下图所示:



设计要点

1、客户端和服务器连接

1.1 与标准的协议相同，客户端向服务器发起 **TCP** 链接，当三次握手完成后，客户端向服务器发送 **TLS** 请求 **client hello**。内容如下：

```
TLSPlaintext:
|type: Uint8(0x16) == handshake
|legacy_record_version: Uint16(0x0303) == TLS12
|length: Uint16(0x022c) == 556
|fragment:
    Handshake:
    |msg_type: Uint8(0x01) == client_hello
    |length: Uint24(0x000228) == 552
    |msg:
        ClientHello:
        |legacy_version: Uint16(0x0303) == TLS12
        |random: bbd4b039958d9f71eb36... (len=32)
        |legacy_session_id: 0d26cf8c69a9fe253fc3... (len=32)
        |cipher_suites: [Uint16(0x1303)]
        |legacy_compression_methods: [Uint8(0x00)]
        |extensions:
            [Extension:
            |extension_type: Uint16(0x002b) == supported_versions
            |extension_data:
                SupportedVersions:
                |versions: [Uint16(0x0304), Uint16(0xbaba)],
             Extension:
            |extension_type: Uint16(0x000a) == supported_groups
            |extension_data:
                NamedGroupList:
                |named_group_list: [Uint16(0x001d), Uint16(0x0100)],
             Extension:
            |extension_type: Uint16(0x000d) == signature_algorithms
            |extension_data:
                SignatureSchemeList:
                |supported_signature_algorithms: [Uint16(0x0809), Uint16(0x080a), Uint16(0x080b), Uint16(0x0804), Uint16(0x0805), Uint
16(0x0806), Uint16(0x0403), Uint16(0x0503), Uint16(0x0603), Uint16(0x0807), Uint16(0x0808)],
             Extension:
            |extension_type: Uint16(0x0033) == key_share
            |extension_data:
                KeyShareClientHello:
                |client_shares:
                    [KeyShareEntry:
                    |group: Uint16(0x001d) == x25519
                    |key_exchange: 852223726f7c521a25ea... (len=32),
                     KeyShareEntry:
                    |group: Uint16(0x0100) == ffdhe2048
                    |key_exchange: c3cf3490adeef60f374b... (len=384)]]
```

其中 **legacy_version** 根据 **RFC** 文档需要写 **TLS1.2**，还必须包括 **legacy_session_id**、**random** 以及 **extension**，**Extension** 含有 **supported_version**、**supported_groups**、**signatureschemelist** 以及 **key_shared**（含选择的公钥）。

1.2 服务器收到之后，需选择支持的最高版本、密钥分发算法和选择的公钥、加密签名算法、以及 **random** 和 **session_id** 回复 **server Hello**，算出自己前主密钥。紧接着使用自己的选择的加密方式加密发送一个 **Encryption_Entension** 报文，接着服务器加密发送 **CA** 证书与数字签名，然后等待客户端的回复 **Finshed**。相关实例：

选择相关套件：

```
TLSPlaintext:
|type: Uint8(0x16) == handshake
|legacy_record_version: Uint16(0x0303) == TLS12
|length: Uint16(0x01da) == 474
|fragment:
    Handshake:
    |msg_type: Uint8(0x02) == server_hello
    |length: Uint24(0x0001d6) == 470
    |msg:
        ServerHello:
        |legacy_version: Uint16(0x0303) == TLS12
        |random: f7866135f34bfddc153c... (len=32)
        |legacy_session_id_echo: 0d26cf8c69a9fe253fc3... (len=32)
        |cipher_suite: Uint16(0x1303) == TLS_CHACHA20_POLY1305_SHA256
        |legacy_compression_method: Uint8(0x00) == 0
        |extensions:
            [Extension:
            |extension_type: Uint16(0x002b) == supported_versions
            |extension_data:
                SupportedVersions:
                |selected_version: Uint16(0x0304) == TLS13,
             Extension:
            |extension_type: Uint16(0x0033) == key_share
            |extension_data:
                KeyShareServerHello:
                |server_share:
                    KeyShareEntry:
                    |group: Uint16(0x0100) == ffdhe2048
                    |key_exchange: d53048c6694cbcc96b97... (len=384)]
```

算出前主密钥:

```
pre_master_shared_key: e0b14444df2cae6456459e86868338958158a0951bc80852217c24cfd8be5fb7dba4a110799f8e7f9929bc0a0378afbe0c
fd3f59fd2db5e9b6e97786a1f7d408898fe78748273043786148064 2cebb067eea97490363d32ddec50c6c5043e6450196d681983913fc2d837ecd9b78
f3780b367e04ffc58ed52097276ab0bd28e42c2e9f54be8ef5ae0977ea917ab1ec5d6f9eab75a672ef74ec7b9421b907de1b286cf8d8425711abb8dff9
dd36aa30e984097a764067ad7ba2b6bfc16679703d7d7c00454261bedefc9fb1cafc181f772f0df349ce588cee44ecde6e38c266c9cf1f68f0dc7d9353
66f4bbed5ebe78837d4c7a4b0f9f43a40e3432ff423d3a150527e8af269676f7db7a5612c96ca14ab6a2781d0448e37abbbbbca302f1c726a8236226c8
8f3a8c13247eaaad126f21d7a115213bd62540739faa0e6396ecd86579841490378ef38af6b7b8f3c774761ed71a0c079f9da8f525910b9efd35066db8
d9389210b49450416b2e996d2e6b7c8d7d0c02296979c66530bed03cd7d9
```

加密发送 **Encrypted_extensions**

```
TLSPlaintext:
|type: Uint8(0x16) == handshake
|legacy_record_version: Uint16(0x0303) == TLS12
|length: Uint16(0x0006) == 6
|fragment:
    Handshake:
    |msg_type: Uint8(0x08) == encrypted_extensions
    |length: Uint24(0x000002) == 2
    |msg:
        EncryptedExtensions:
        |extensions: []
```

加密发送证书:

```
TLSPlaintext:
|type: Uint8(0x16) == handshake
|legacy_record_version: Uint16(0x0303) == TLS12
|length: Uint16(0x036d) == 877
|fragment:
    Handshake:
    |msg_type: Uint8(0x0b) == certificate
    |length: Uint24(0x000369) == 873
    |msg:
        Certificate:
        |certificate_request_context:  (len=0)
        |certificate_list:
            [CertificateEntry:
            |cert_data: 3082035c308202440209... (len=864)
            |extensions: []]
```

加密发送证书的数字签名的验证：

```
<<<   CertificateVerify   >>>

TLSPlaintext:
|type: Uint8(0x16) == handshake
|legacy_record_version: Uint16(0x0303) == TLS12
|length: Uint16(0x0108) == 264
|fragment:
    Handshake:
    |msg_type: Uint8(0x0f) == certificate_verify
    |length: Uint24(0x000104) == 260
    |msg:
        CertificateVerify:
        |algorithm: Uint16(0x0809) == rsa_pss_pss_sha256
        |signature: a180b1ff17d0e0a446b2... (len=256)
```

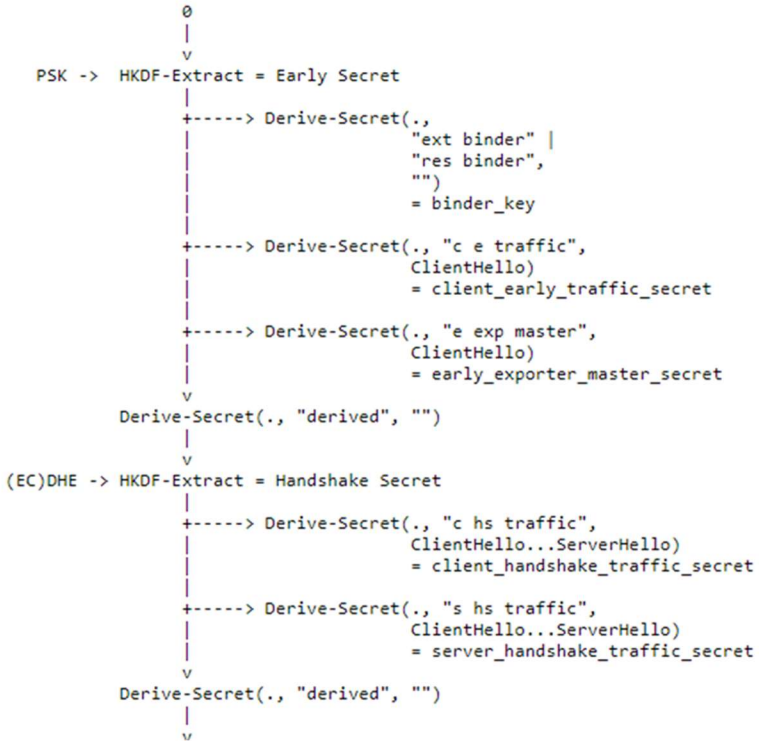
1.3 客户端收到服务器的 **Server Hello** 报文后，计算前主密钥，解密接下来接收到的文件，验证其正确性，若有任何一点有问题，发送警告报文，然后中断此次握手过程，重新建立握手过程。若正确，加密发送 **Finshed** 报文，然后可以加密发送数据。

```
TLSPlaintext:
|type: Uint8(0x16) == handshake
|legacy_record_version: Uint16(0x0303) == TLS12
|length: Uint16(0x0024) == 36
|fragment:
    Handshake:
    |msg_type: Uint8(0x14) == finished
    |length: Uint24(0x000020) == 32
    |msg:
        Finished:
        |verify_data: 3cd9eb9df07b051bf2da... (len=32)
```

1.4 服务器接计算主密钥，然后在收到 **Finshed** 报文后，加密发送 **Finshed** 报文，然后握手成功，可以选择新会话 **tickts** 报文。

2.会话密钥生成

首先，客户端用 **diffie-hellman** 算法以及 **EC** 算法生成公钥，私钥是系统中 **OS.urandom()**里面的随机数，服务器选择了 **DHE** 算法，然后选择随机数 **x**，发送 **pow(g, x)**给客户端，现在双方都已知 **pow(g, xy)mod p** （**g p** 为已知参数），将其作为前主密钥。然后使用 **HKDF_Extract** 以及 **Derive_secret** 算法类加上会话消息的 **MAC** 值，按如下格式生成主密钥、握手密钥、数据传输密钥。



```
                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
```

3.加密解密

加密解密我使用了 **RFC8446** 规定的 **TLS_CHACHA20_POLY1305_SHA256** 加密套件，加解密使用 **aead** 方式，实现方式完全按照 **RFC8439**（**https://tools.ietf.org/html/rfc8439**）文档中规定的加解密参数以及细节。算法概要:

2.5.1. The Poly1305 Algorithms in Pseudocode

```
clamp(r): r &= 0x0ffffffc0ffffffc0ffffffc0fffffff
poly1305_mac(msg, key):
   r = le_bytes_to_num(key[0..15])
   clamp(r)
   s = le_bytes_to_num(key[16..31])
   a = 0  /* a is the accumulator */
   p = (1<<130)-5
   for i=1 upto ceil(msg length in bytes / 16)
      n = le_bytes_to_num(msg[((i-1)*16)..(i*16)] | [0x01])
      a += n
      a = (r * a) % p
      end
   a += s
   return num_to_16_le_bytes(a)
   end
```

4. MAC 和验证

我选择了没有使用 **PSK**，而是使用 **(EC)DHE** 和基于证书的认证。所以 **Server** 会发送 **Certificate** 和 **CertificateVerify** 消息。由于在所有的握手中，**Server** 必须在 **ServerHello** 消息之后立即发送 **EncryptedExtensions** 消息。这是在从 **server_handshake_traffic_secret** 派生的密钥下加密的第一条消息。

注:

Certificate：用于认证的证书和链中任何支持的证书。

CertificateVerify: 根据 **Transcript-Hash(Handshake Context, Certificate)**的值得出的签名

Finished: 根据 **Transcript-Hash(Handshake Context, Certificate, CertificateVerify)**的值得出的 **MAC** 。

我使用的数字签名的计算内容：**1.**由八位字节 **32(0x20)**组成的字符串重复 **64** 次，**2.**上下文字符串 **3.**用作分隔符的单个 **0** 字节 **4.**要签名的内容 ；这里我使用的签名算法是 **SHA256**，因为与 **chahca20plog1305** 是一个组件，设计这个结构目的是为了防止对先前版本的 **TLS** 的攻击，其中 **ServerKeyExchange** 格式意味着攻击者可以获得具有所选 **32** 字节前缀 **(ClientHello.random)**的消息的签名。 **Server** 签名的上下文字符串是 **"TLS 1.3**，**Server CertificateVerify"**。当客户端收到后，进行验证 **CertificateVerify** 消息的签名字段。验证过程的输入：**1**、数字签名所涵盖的内容 **2**、在关联的证书消息中找到的最终实体证书中包含的公钥 **3**、在 **CertificateVerify** 消息的签名字段中收到的数字签名。

代码细节介绍

1.密钥协商是用的 **FFDHE**（**Finite Field Diffie-Hellman Ephemeral Parameters**），使用了 **RFC7919** 中定义的相关参数 **g**、**p**。私钥是从 **os.urandom()**里面获取的随机数。然后生成公钥。以及直接调用 **python** 包 **cryptography.hazmat.primitives.asymmetric.x25519**，使用 **x25519** 进行密钥分发。由于我只实现了 **TLS_CHACHA20_POLY1305_SHA256** 的

加密算法。随机数和 **seesion_ID** 我是直接调用 **python** 包 **secrets** 里面的函数 **token_bytes(32)**，生成 **32** 位长的随机数，但是 **token_bytes** 是直接调用 **os.urandom(32)**得到的，所以这个随机数和 **Seesion_ID** 跟前面私钥的生成一样，是取自系统 **urandom**。

2.每一种选项集合，我都使一个特定的类表示，每个类有一个参数成员 **_size** 用来表示每个成员在传输过程中应占用的大小。

```
enum {
    client_hello(1),
    server_hello(2),
    new_session_ticket(4),
    end_of_early_data(5),
    encrypted_extensions(8),
    certificate(11),
    certificate_request(13),
    certificate_verify(15),
    finished(20),
    key_update(24),
    message_hash(254),
    (255)
} HandshakeType;
```

```
@Type.add_labels_and_values
class HandshakeType(Type):
    """..."""
    client_hello = Uint8(1)
    server_hello = Uint8(2)
    new_session_ticket = Uint8(4)
    end_of_early_data = Uint8(5)
    encrypted_extensions = Uint8(8)
    certificate = Uint8(11)
    certificate_request = Uint8(13)
    certificate_verify = Uint8(15)
    finished = Uint8(20)
    key_update = Uint8(24)
    message_hash = Uint8(254)
    _size = 1
```

3.由于 **TLS1.2** 的广泛使用，部分 **TLS** 不支持的加密算法，在实际的操作中还是存在，例如：

```
Cipher Suites Length: 34
Cipher Suites (17 suites)
    Cipher Suite: Reserved (GREASE) (0x2a2a)
    Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
    Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
    Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9)
    Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca8)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
    Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
    Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
    Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
    Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
```

所有的数据来源于最新版的 **Chrome(Version 70.0.3538.110 (Official Build) (64-bit))**浏览器和 **gmail.com** 之间的通信。因为两者都支持 **TLS1.3**，通信

过程中是使用的 **TLS1.3** 进行通信。由于以前 **TLS** 协议中的密钥交换算法仅仅只有 **EC**，所以现在大多数也较常用 **EC**（椭圆曲线）。

4.代码文件说明

Main.py 主文件入口函数

Server.py 是服务器的主文件

Client.py 是客户端的主文件

Protocal 文件中包含 **TLS** 握手的主要数据结构和函数实现，以及 **keyexchange** 里面包含于密钥协商相关的实现函数与类

Utilization 里面是为了帮助实现功能的各种函数以及类，包括密码算法的实现以及加解密算法。还有数据结构的转换函数类。

结果

程序运行测试通过。由于时间原因，还没有使用 **wireshark** 进行包的解析测试，以及 **RFC** 文档中部分细节函数并没有完全实现，很多加解密算法不支持，还没有完成多线程，现在还不能进行双方同时发送接收信息，使用了 **sleep** 函数进行有关同步，使程序有卡顿。总之，还有好多要实现。效果：

参考链接：

https://github.com/halfrost/Halfrost-Field/blob/master/contents/Protocol/TLS_1.3_Handshake_Protocol.md

https://blog.csdn.net/mrpre/