# *Module 02: Key Exchange and the TLS 1.3 Protocol*

## *Week-05: Implementing the TLS 1.3 Protocol*

*Benjamin Dowling, Jan Gilcher, Kenny Paterson, and Sikhar Patranabis*

*benjamin.dowling@inf.ethz.ch, jan.gilcher@inf.ethz.ch*
*kenny.paterson@inf.ethz.ch, sikhar.patranabis@inf.ethz.ch*

*October 2020*

Hi all! Welcome to the second Information Security Lab for this module. This lab is part of the Module-02 on *Key Exchange and TLS 1.3*.

## *Instructions for Online Lab*

We begin with a few pointers and instructions. As you are aware, the lab sessions will be held online via Zoom. We ask that all students join the Zoom session via the following link:

`https://ethz.zoom.us/j/91450896871`

The first 30-ish minutes of the Zoom session will consist of a general presentation and discussion about the lab objectives and evaluation criteria. Afterwards, you may choose to stay on in the lab session to discuss specific problems you might be having with the lab. You are also welcome to post your questions and to join ongoing discussions on the Moodle forum at the following link:

`https://moodle-app2.let.ethz.ch/course/view.php?id=13172`

During the Zoom session, we will be hosting "break-out rooms". These are meant to facilitate one-on-one discussions with one of the TAs. During such discussions, you can share your screen and bring to our attention specific implementation issues that you might be facing. Please be aware that when you are sharing a screen with the session, you are responsible for the content that is presented to any other students in the breakout. In other words, *please be mature*.

## *Overview*

This lab serves as an opportunity to investigate and implement a (streamlined) version of one of the most important cryptographic protocols in use today - the Transport Layer Security Protocol, version 1.3 (throughout the lab we will refer to this simply as TLS 1.3). In this lab, we will use the symmetric and asymmetric cryptoprimitives you have seen in previous weeks, and use them to implement various cryptographic primitives specific to TLS 1.3. We will also be helping to implement client functions for the

Handshake Protocol - an “authenticated key exchange" (AKE) protocol - and the Record Protocol - a “secure channel” protocol.

As described by the TLS 1.3 RFC [1], the handshake protocol “*authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.*”

Afterwards, the record protocol “*uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.*”

Please note that we expect you to use Python-3 for this lab. We will not accept submissions/solutions coded in any other language, including older versions of Python. *So please make sure that your submissions are Python-3 compatible.*

The focus of this lab is on cryptographic API, as well as engaging with formal specification documents and cryptographic documentation. We hope that this lab will give you insight into the experience of coding real-world applications and applied cryptography.

## *Getting Started*

In case you haven’t already done this last week, for this lab you will need to install an elliptic-curve library using `pip3 install tinyec`, and modern AEAD libraries using PyCryptodome (`pip3 install pycryptodome` or `pip3 install pycryptodomex` depending on whether PyCrypto has already been installed / if you want to keep both). Note that if you encounter errors during the installation, it may be worth trying to `pip3 uninstall pycrypto`. Similarly to the last module, we are also providing a virtual machine (VM) image that has the necessary dependencies pre-installed and ready to use.

The link to access and download the VM is available on Moodle. Below we provide you with a simple set of instructions on how to use this VM:

- Boot the VM using your favorite virtualization software. For example, you can download and use the Oracle VM Virtualbox from the url below:

  `https://www.virtualbox.org/`

- Login using the root password `isl`
- At this point, you can run any Python script(s) that are required for this assessment.

Note that when automatically evaluating your code, we will use this same VM and the same build environment. So using the VM would additionally allow you to pre-test your submission in an environment resembling our automated testing environment.

Before we begin, we recap the cryptographic background and notation that we will be using throughout this lab sheet.

## *Background and Notations*

- $(x, X = g^x) \leftarrow_R \mathsf{DH.KeyGen}(\lambda)$: denotes the Diffie-Hellman key generation algorithm. It takes as input a security parameter $\lambda$, and outputs a secret/public Diffie-Hellman pair $(x, X = g^x)$.
- $(g^{xy}) \leftarrow \mathsf{DH.DH}(x, Y)$: denotes the Diffie-Hellman computation algorithm. It takes as input a Diffie-Hellman secret value $x$ and a Diffie-Hellman public value $Y$, and outputs a shared secret $g^{xy}$.

  For all implementations in this lab, we will be using elliptic-curve Diffie-Hellman to perform DH operations, imported from **tinyec**. For greater readability and ease of exposition, we will be using the exponential notation for Diffie-Hellman throughout i.e we write $g^x$ to represent the scalar multiplication $[x]g$ where $g$ is a point on an elliptic curve.

- $Y \leftarrow \mathsf{H}(X)$: denotes a hash function. It takes as input an arbitrary-length bit-string $X$ and outputs a bit-string of some fixed length $Y$.

  For all implementations in this lab, we will exclusively be using either SHA256 or SHA384, imported from PyCryptodome **Crypto.Hash** [2].
- $k' \leftarrow \mathsf{KDF}(r, s, c, L)$ is a key derivation function. It takes as input some randomness $r$, some (optional) salt $s$, some context input $c$ and an output length $L$, and outputs a key $k'$.

  In this lab, we will be using a key derivation function $\mathsf{KDF}$ (specifically, HKDF, which is implemented in `tls_crypto` using HMAC from **Crypto.Hash** [2]) to derive the symmetric keying material as well the output shared symmetric keys.

- $(\mathsf{sk}, \mathsf{vk}) \leftarrow_R \mathsf{SIG.KeyGen}(\lambda)$: denotes the key generation of a digital signature scheme SIG. It takes as input a security parameter $\lambda$, and outputs a signing key $\mathsf{sk}$ and a verification key $\mathsf{vk}$.
- $\sigma \leftarrow_R \mathsf{SIG.Sign}(\mathsf{sk}, \mathsf{msg})$: denotes the signing algorithm of SIG. It takes as input a signing key $\mathsf{sk}$ and a message $\mathsf{msg}$, and outputs a signature $\sigma$.
- $\{0, 1\} \leftarrow \mathsf{SIG.Verify}(\mathsf{vk}, \mathsf{msg}, \sigma)$: denotes the verifying algorithm of SIG. It takes as input a verifying key $\mathsf{vk}$, a message $\mathsf{msg}$, and a signature $\sigma$ and outputs 0 or 1.

  Again, for correctness we need that for all $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{SIG.KeyGen}(\lambda)$, we have:

  $$\mathsf{SIG.Verify}(\mathsf{vk}, \mathsf{msg}, \mathsf{SIG.Sign}(\mathsf{sk}, \mathsf{msg})) = 1.$$

  In this lab, we will be using ECDSA [3] or RSA [4] signatures to instantiate our digital signature scheme, imported from **Crypto.Signatures**.

- $\tau \leftarrow \mathsf{MAC}(k, \mathsf{msg})$: denotes the message authentication algorithm MAC. It takes as input a symmetric key $k$ and a message $\mathsf{msg}$, and outputs a MAC tag $\tau$.

  In this lab, we will be using HMAC to instantiate our MAC algorithm, imported from **Crypto.Hash** [2].

- $k \leftarrow_R \mathsf{AEAD.KeyGen}(\lambda)$: denotes the key generation algorithm of an authentication encryption scheme with associated data AEAD. It takes as input a security parameter $\lambda$ and outputs a symmetric key $k$.
- $\mathsf{ctxt} \leftarrow \mathsf{AEAD.Enc}(k, nonce, ad, \mathsf{msg})$: denotes the encryption algorithm of AEAD. It takes as input a symmetric key $k$, a nonce $nonce$, some associated data $ad$ and a message msg, and outputs a ciphertext ctxt
- $\{\mathsf{ctxt}, \perp\} \leftarrow_R \mathsf{AEAD.Dec}(k, nonce, ad, \mathsf{ctxt})$: denotes the encryption algorithm of AEAD. It takes as input a symmetric key $k$, a nonce $nonce$, some associated data $ad$ and a ciphertext ctxt, and outputs either a plaintext message msg or an error symbol $\perp$.

  For correctness we need that for all $k \leftarrow \mathsf{AEAD.KeyGen}(\lambda)$, all nonces $nonce$, for all associated data $ad$ and all messages msg, we have:

$$\mathsf{msg} = \mathsf{AEAD.Dec}(k, nonce, ad, \mathsf{AEAD.Enc}(k, nonce, ad, \mathsf{msg}))$$

  In this lab, we will be using `AES_128_GCM` [4], `AES_256_GCM` [4] and `CHACHA20_POLY1305` [5] to instantiate our AEAD schemes, imported from **Crypto.Cipher** [4].

*Limitations*

Unfortunately, you will not be working on a full TLS 1.3 implementation, but a significantly streamlined and "bare-bones" variant of TLS 1.3. This implementation we have provided diverges from the specification in a number of ways (not an exhaustive list):

- We do not capture the TLS state machine at all - our implementation instead advances linearly depending on the input and processed functions.
- We do not capture the TLS alert protocol in our implementation. As a result, the server implementation will not send alert messages in response to some failure to parse a message or verify an authentication value. It will just close the connection.
- We do not capture the majority of TLS extensions listed in the TLS RFC, such as ServerNameIndication. We only use extensions for signature negotiation, ECDHE group negotiation, and version negotiation.
- We do not capture the use of ChangeCipherSpec, sent for compatability purposes.
- Since we only exchange a single message between the client and server (and vice-versa), we do not capture KeyUpdate mechanisms within the Record Layer. Similarly, our implementation does not send Closure Alerts, which protect against truncation attacks.
- Since we know exactly which Diffie-Hellman groups that the server supports ahead of time, our server implementation does not support HelloRetryRequest in the event of failing to find sufficient information to proceed with a TLS 1.3 handshake.

The point that we are making here is that this is *not* a full TLS implementation, and we would not recommend its usage in the wild.

## *Implemented Functions*

Before we begin, it is worth highlighting some code that has already been provided to you in the skeleton file. These are essentially functions based on the tinyEC library to implement elliptic curve cryptography [7]. These would assist you in generating a (scalar, point) key-pair, and execute basic scalar multiplication operations on the curve. If you are unfamiliar with last week's lab, you may wish to read the documentation [7], or refer to the Additional Listings section at the end of the sheet for an example of how to utilise tinyec.

## *High-Level API*

Let's begin by looking at the highest level of our TLS implementation. This is not really a part of TLS, but instead a simple sockets implementation that allows network communication. The TLS specific parts of these functions is about accessing the high-level TLS API for:

- handshake functions for both client and server
- record layer functions for both client and server.

In what follows, we will be focusing on the client-specific functions, since that is what you will be implementing for your assessment. Below we give a listing for the "simple_client.py" file, and discuss what this does.

Listing 1: Simple Client Socket

```
import socket
import tls_application
import tls_constants

def client_socket():
        s = socket.socket()
        host = socket.gethostname()
        #host = '18.216.1.168'
        port = 1189
        s.connect((host, port))
        client = tls_application.TLSConnection(tls_constants.CLIENT_FLAG)
        tls_client_hello = client.begin_tls_handshake()
        s.send(tls_client_hello)
        server_messages = s.recv(2048)
        client_messages = client.finish_tls_handshake_client(server_messages)
        s.send(client_messages)
        client_enc_message = client.send_enc_message("challenge".encode())
        s.send(client_enc_message)
        server_enc_message = s.recv(1024)
        ptxt_message = client.recv_enc_message(server_enc_message)
        print(ptxt_message.decode('utf-8'))
        s.close()

client_socket()
```

This is a fairly simple function, and we can go through how it works together. The `client_socket` function begins by creating a socket object, allowing network communication. You can initialise the host as either a local host, or a remote host with address '18.216.1.168'. This remote host is a *complete* (according to the lab sheet) `simple_server` running on an Amazon Web Service instance, for you to ping with messages and see how well your implementation can interoperate with our implementation. Afterwards, the `simple_client` function connects to the (*host*, *port*) pair provided and intialises a TLSConnection object with a CLIENT_FLAG - you can look through `tls_constants.py` if you are interested in seeing what that flag looks like.

The client uses the TLSConnection function `begin_tls_handshake` - the output of which is `tls_client_hello`, which is sent to the server via the pre-established socket. The client receives "server_messages" as a response, which is processed via "finish_tls_handshake_client", which outputs the last flight of the client messages, which is sent to the server. Now the client can begin sending and receiving protected messages via the Record Layer. In this example, the client will always send the first message ("challenge") while the sender will always wait to receive this (valid) message, before it's response ("response"), which the client will print, and afterwards, close the socket connection.

Now we have seen how this API will be used, let us focus on the main tasks that you will have to complete for this lab.

*Overview*

In this lab, you will be implementing a series of modular TLS cryptographic primitives. You will have access to a folder containing a series of python files implementing various aspects of the TLS 1.3 protocol, and test vectors for testing your implementation. We list these below and describe on a high-level what each file is contributing to our TLS implementation:

- `simple_client.py`: This file creates sockets and manages networking tasks for a client TLS instance.
- `simple_server.py`: This file creates sockets and manages networking tasks for a server TLS instance. We separate this files so you can run a server locally and have your client interact with it.
- `test_tls_handshake.py`: This file will allow you to run unit tests and see how well your implementation of the tls-specific client handshake functions match ours
- `tls_application`: This file contains the API that connects the high-level functions contained in simple_client.py and simple_server.py to the appropriate Handshake and Record functions contained in tls_handshake and tls_record, respectively.
- `tls_crypto`: This file contains the tls-specific cryptographic functions.
- `tls_error`: This file contains some basic errors that may occur during the execution of a TLS Handshake or TLS Record Layer protocol.
- `tls_extensions`: This file contains functions to manage the preperation and negotiation of TLS extensions sent during the ClientHello and ServerHello messages.

- **tls_handshake**: This file contains handshake functions for both client and server handshake functions - some of which you will be implementing for this assessment.
- **tls_psk_functions**: This file contains some PSK functionality that you will be implement for both clients and servers. In addition you will be implementing the full TLS key schedule.
- tls_record: This file contains high-level API for preparing both plaintext and encrypted TLS record packets.

In what follows, you will be expected to be able to support the following cryptographic options:

- Ciphersuites: TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256. This means that when you use hash functions or AEAD schemes, you will need to be able to distinguish between use of SHA256 or SHA384, and AES_128_GCM, AES_256_GCM, or CHACHA20_POLY1305 respectively. All functions that you implement that requires this distinct behaviour will be given csuite as input - an integer representation of the negotiated ciphersuite, which will allow you to distinguish which algorithms you require. The various csuite values are defined in tls_constants.py, and we recommend you look through this file.
- Elliptic-Curve Diffie-Hellman (ECDH) groups: You will required to support SECP256R1, SECP384R1 or SECP521R1. Similarly, all functions that you implement that requires distinct behaviour depending on the negotiated group will be given neg_group as input - an integer representation of the negotiated group. The various neg_group values are defined in tls_constants.py.
- Signature schemes: You will be required to support RSA_PKCS1_SHA256, RSA_PKCS1_SHA384, and ECDSA_SECP384R1_SHA384. As before, all functions that you implement that requires distinct behaviour depending on the negotiated signature scheme will be given signature_algorithm as input - an integer representation of the negotiated signature scheme. The various neg_group values are defined in tls_constants.py.

With our overview done, let us take a closer look at tls_dandshake.py, and describe the expected API and operations for each.

## *TLS Handshake Functions*

In this portion of the lab you also be implementing a series of Client Handshake functions. Specifically, the function that creates the `ClientHello` message, the function that parses the `ServerHello` message, the function that verifies the `ServerCertificateVerify` message, and finally, the function that verifies the `ServerFinished` message. All these functions are defined in `tls_handshake.py`

Before we begin, we should point to what a TLSPlaintext packet looks like. Consider the following from the TLS RFC [1], from Section 5.1:

```
struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

type: The higher-level protocol used to process the enclosed fragment.

legacy_record_version: MUST be set to 0x0303 for all records generated by a TLS 1.3 implementation other than an initial `ClientHello` (i.e., one not generated after a HelloRetryRequest), where it MAY also be 0x0301 for compatibility purposes. This field is deprecated and MUST be ignored for all purposes. Previous versions of TLS would use other values in this field under some circumstances.

length: The length (in bytes) of the following TLSPlaintext.fragment. The length MUST NOT exceed $2^{14}$ bytes. An endpoint that receives a record that exceeds this length MUST terminate the connection with a "record_overflow" alert.

fragment: The data being transmitted. This value is transparent and is treated as an independent block to be dealt with by the higher- level protocol specified by the type field.

Note that ContentType is a integer represented as a single byte in big-endian (network) order:

```
enum {invalid(0), change_cipher_spec(20), alert(21),
       handshake(22), application_data(23), (255)} ContentType;
```

In `tls_handshake.py`, we define a function `attach_handshake_header` that, given an integer `msg_type` and a series of bytes `msg`, will concatenate this handshake header to the beginning of `msg`. Similarly, we have defined a function `process_handshake_header` that, given an integer `msg_type` and a series of bytes `msg`, strips the header from the message `msg`. If these messages are sent encrypted, the plaintext is then given another wrapper:

```
struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;
```

Finally, the RecordLayer header is then attached to the message:

```
struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

You may recall the second part from the AD field used in the AEAD scheme you implemented earlier. While interesting, the Record Layer is out of scope for your assessment, so let's circle back around to the function `tls_13_client_hello`.

### *TLS Client Hello*

Listing 2: TLS Client Hello

```python
def tls_13_client_hello(self):
        raise NotImplementedError()
```

In this function, you will be implementing the creation of a valid TLSPlaintext `ClientHello` message. This function should take no additional inputs (beyond *self*) and output `client_hello_msg` - a series of bytes comprising the ClientHello message. Section 4.2.1 of the TLS RFC [1] describes the structure of a TLS ClientHello message, consider the following below:

Structure of this message:

```
struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<8..2^16-1>;
} ClientHello;

uint16 ProtocolVersion;
opaque Random[32];
uint8 CipherSuite[2];    /* Cryptographic suite selector */
```

Recall that the $< X..Y >$ notation means that the field must be preceded by a length-encoding, large enough to represent $Y$ in big-endian (network) order. The `Random` and `legacy_session_id` structures are straightforward - according to the RFC, both are 32-bytes of randomness. The `Random` structure acts as a nonce, so it should be generated via a "secure random number generator". We have imported **get_random_bytes** for you to use.

Ciphersuites is a list of the symmetric cipher options supported by the client - the values for the ciphersuites you are expected to support are given in `tls_constants.py`. Each "ciphersuite" in this list is a 2-byte representation of the integer value given in `tls_constants.py`, in big-endian order. To make this simpler for you, we have already initialised the client and server with the list of ciphersuites they will support, in *self*.`csuites` (where *self*.`csuites` is a tuple of integers). you will have to encode each integer as a 2-byte big-endian representation. Don't forget to add the length encoding!

The `legacy_compression_methods` " MUST contain exactly one byte, set to zero, which corresponds to the "null" compression method in prior versions of TLS." Remember that the <> notation means that you must still include the length-encoding field.

In our TLS implementation, we support the following extensions:

- `supported_versions`
- `supported_groups`
- `key_share`
- `signature_algorithms`

For the structure of the extension fields, consider the following from Section 4.2 of the TLS RFC [1]:

```
struct {
     ExtensionType extension_type;
     opaque extension_data<0..2^16-1>;
} Extension;
```

Each extension, like a Handshake or RecordLayer header, begins with a `extension_type` identifying the extension, and a field encoding the length of the `extension_data`.

The file `tls_extensions.py` contains functions for preparing each extension:

- `tls_extensions.prep_support_vers_ext`(*self*.`extensions`)
- `tls_extensions.prep_support_groups_ext`(*self*.`extensions`)
- `tls_extensions.prep_keyshare_ext`(*self*.`extensions`)
- `tls_extensions.prep_signature_ext`(*self*.`extensions`)

Each takes *self*.`extensions` as input (which we have already initialised in our implementation), and outputs the given extension in byte format.

The only exception is `tls_extensions.prep_keyshare_ext`, which will return two outputs: The extension itself, and a dictionary of ECDH secret values, indexed by the integer representation of each supported group. You should save this dictionary to *self*.`ec_sec_keys`, as you will need to extract one of these later in the handshake. We recommend you look through the `tls_extensions.py` file to determine what each of these extensions look like, and how negotiation of such extensions occur.

In standard TLS implementations, the ordering of Extensions should not matter. Unfortunately, this is a place where our implementation differs from a standard TLS implementation: **always** place the `supported_groups` extension *before* the `key_share` extension. Don't forget to add the Length encoding once you have concatenated all extensions together!

Now you have created each of the fields in the `ClientHello`, simply concatenate the fields (in the correct order!) and add the appropriate TLSPlaintext header. Your implementation will also need to update *self*.`transcript` to include the `ClientHello` message, in order to compute the transcript hash in later stages of the Handshake.

This portion of the assessment is intended to familiarise you with TLS packet structure, reading specifications and correctly aligning expected inputs for various pre-established functions. Now we have finished discussing how to create a ClientHello message, let's continue onto the second part of our Client Handshake functions for you to implement: `tls_13_process_server_hello`.

### *TLS Process Server Hello*

Listing 3: TLS Process Server Hello

```
def tls_13_process_server_hello(self, shelo_msg):
        raise NotImplementedError()
```

As the name suggests, in `tls_13_process_server_hello` you will be implementing a function that parses the ServerHello message, and extracts the *ciphersuite*, *version*, *ECDH group* and *ECDH keyshare* that the server has negotiated. This means implementing code that can parse variable-length extensions. In addition, your function will also use previously established API to compute secret ECDH values, and derive a series of secrets. This function should take inputs *self*, `shelo_msg` (where *self* is the current state, and `shelo_msg` a series of byte comprising the ServerHello message, and return no output. Section 4.1.3 of the TLS RFC [1] describes the structure of a TLS ServerHello message, consider the following below:

```
Structure of this message:

struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id_echo<0..32>;
    CipherSuite cipher_suite;
    uint8 legacy_compression_method = 0;
    Extension extensions<6..2^16-1>;
} ServerHello;
```

Recall again that the $< X..Y >$ notation means that the field must be preceded by a length-encoding, large enough to represent $Y$ in big-endian (network) order. The `Random` is generated in exactly the same way as in `ClientHello` - by generating 32 random bytes. However, `legacy_session_id` should by an echo of the client's recently sent legacy_session_id - you can check for yourself that our implementation does this in tls_13_process_client_hello and tls_13_server_hello.

After the legacy_session_id comes the single CipherSuite that the server has negotiated - your implementation should set *self*.csuite to the *integer* value of this field. legacy_compression_method is a single byte that indicates that the server's choice of compression method. In TLS 1.3 this must be the "null" compression method - hence legacy_compression_method = 0.

Finally, you must process the extensions that the Server has sent. These will be following extensions:

- supported_groups
- supported_versions
- key_share

You'll notice that one of the extensions sent in the ClientHello is not present, specifically signature_algorithms. Why would the Server not need to send this extension?

As with CipherSuite, supported_versions and supported_groups will each contain a single Group and Version that the Server has negotiated. *Unlike* CipherSuite, you will need to parse the extension identifier and extension length. For each, your implementation should set *self*.neg_group (respectively, *self*.neg_version) to the *integer* value of this field.

The key_share extension has a slightly more complex format, which we'll examine below. Consider the following from Section 4.2.8 (KeyShare) from the TLS RFC [1]:

```
struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;
```

Keep in mind that this KeyShareEntry is the extension_data sent in an Extension. KeyShareEntry contains the NamedGroup and the confusingly titled key_exchange fields. NamedGroup is a 2-byte representation of integer value indicating the NamedGroup, defined in the TLS RFC. For our implementation, these integer values are defined in tls_constants.py, for instance SECP256R1_VALUE = 0x0017. Here key_exchange is the Diffie-Hellman keyshare. First, note the variable-length notation for key_exchange. Next, we turn to the RFC to see how the key_exchange field is formatted, Section 4.2.8 :

```
struct {
    uint8 legacy_form = 4;
    opaque X[coordinate_length];
    opaque Y[coordinate_length];
} UncompressedPointRepresentation;
```

Our TLS implementation follows this format: A single byte, followed by the *X* and *Y* co-ordinates of the Elliptic Curve Diffie-Hellman key share. To make this simpler, we have included two functions in `tls_crypto.py` to help you convert between tinyEC elliptic curve Point objects and bytes:

- `convert_ec_pub_bytes(ec_pub_key, group_name)`
- `convert_x_y_bytes_ec_pub(pub_bytes, group_name)`

Straightforwardly, `convert_ec_pub_bytes(ec_pub_key, group_name)` takes as input a tinyEC EC Point object, and a group name and return a series of bytes.

`convert_x_y_bytes_ec_pub(pub_bytes, group_name)` takes a series of bytes and a group name, and returns a tinyEC Point object. For consistency, the `group_name` is the integer value used to indicate `NamedGroups` in `KeyShare` extensions.

Your implementation, when parsing `KeyShareEntry`, should convert the `NamedGroup` to an integer, and then use `tls_crypto.convert_x_y_bytes_ec_pub` to create a tinyEC Point object. After, the function should set *self*.`ec_pub_key` to this tinyEC Point object.

Once you have finished processing the `ServerHello` message, there are only three parts left to complete for this function:

1. Update *self*.`transcript`, by concatenating the `shelo_msg` to the end.
2. Compute the Diffie-Hellman secret value, using:
   - `tls_crypto.ec_dh(ec_sec_key, ec_pub_key)`, which takes as input an integer `ec_sec_key` and a tinyEC Point `ec_sec_key`, and returns a tinyEC Point `ec_secret_point`. For this part, you will need to extract the ECDH secret value from *self*.`ec_sec_keys`, set during `tls_13_client_hello`. Recall what type of structure *self*.`ec_sec_keys`, and how to extract values from that structure. You can examine `tls_constants.py` `GROUP_FLAGS` to see how we convert integers used in TLS to indicate groups, to strings used in tinyEC to indicate groups, in order to make the interface on the `tls_handshake.py` level more uniform for you.
   - `tls_crypto.point_to_secret`(`ec_point`, `group`), which takes as input a tinyEC Point `ec_point`, and an integer `group` and returns a series of bytes `ecdh_secret`
3. Extract and derive these secrets, according to the key schedule on the next page:
   - *self*.`early_secret`, using `tls_crypto.tls_extract_secret`.
   - *self*.`handshake_secret`, using `tls_crypto.tls_derive_secret` and `tls_crypto.tls_extract_secret`.
   - *self*.`local_hs_traffic_secret`, using `tls_crypto.tls_derive_secret`
   - *self*.`remote_hs_traffic_secret`, using `tls_crypto.tls_derive_secret`
   - *self*.`master_secret`, using `tls_crypto.tls_derive_secret` and `tls_crypto.tls_extract_secret`.

When **None** appears in the key schedule, use this input literally, i.e. `early_secret` = `tls_crypto.tls_extract_secret`(*self*.`csuite`, None, None).

```
       None
        |
        v
None   ->  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
None    -> 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
```

*TLS Process Server Certificate Verify*

Listing 4: TLS Process Server Certificate Verify

```
def tls_13_process_server_cert_verify(self, verify_msg):
    raise NotImplementedError()
```

In this function, you will be processing the first server authentication message, ServerCertificateVerify. On a high-level, the message is simply a signature over a hash of the current transcript, which you will verify using the server public-key that can be extracted from the ServerCertificate message. This function should take inputs *self*, `verify_msg` (where *self* is the current state, and `verify_msg` a series of bytes comprising the ServerCertificateVerify message, and return no output.

Much like previous messages, you will first need to process the handshake header via *self*.`process_handshake_header`, which takes as input an integer representing the expected handshake (see `tls_constants` for definitions of handshake types) and the handshake message itself (given in bytes).

Consider the following from Section 4.4.3 (Certificate Verify) of the TLS RFC [1]:

Structure of this message:

```
struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;
```

`SignatureScheme algorithm` here refers to the 2-byte representation of the integer value corresponding to the Signature scheme that the Server used to create the signature. You can see now why the Server did not need to send an extension indicating which signature scheme was negotiated, as it is indicated within this CertificateVerify message instead. Again, take note of the variable-length notation here, and recall what that implies for the structure of the message. Once your implementation has extracted the signature from the CertificateVerify message, there are only four steps that follow:

1. Extract the public-key from the certificate that the Server sent in an earlier message. To help you, our implementation has saved the Server Certificate as a string in *self*.`server_cert_string`. In addition, we have provided the following functions:
   - `tls_crypto.get_rsa_pk_from_cert`, which takes as input a certificate string, and outputs an RSA Public Key object
   - `tls_crypto.get_ecdsa_pk_from_cert`, which takes as input a certificate string, and outputs an ECDSA Public Key object
2. Create a transcript hash from the currently maintained *self*.`transcript`. Your previous implementation of `tls_transcript_hash` will help you here.
3. Verify the signature. Your previous implementation of `tls_verify_signature` will help you here.

4. If the signature verifies correctly, update *self*.`transcript` with the CertificateVerify message you just processed.

### *TLS Process Finished*

Listing 5: TLS Process Finished

```
def tls_13_process_finished(self, fin_msg)
raise NotImplementedError()
```

In this function, you will be processing the second authentication message, the Finished message. Your implementation should be general enough for both the client and server to use this function. On a high-level, the message is simply a MAC tag over a hash of the current transcript, which was computed using the `finished_key`.

Much like previous messages, you will first need to process the handshake header via *self*.`process_handshake_header`, which takes as input an integer representing the expected handshake (see `tls_constants` for definitions of handshake types) and the handshake message itself (given in bytes).

Consider the following from Section 4.4.4 (Finished) of the TLS RFC [1]:

Structure of this message:

```
struct {
    opaque verify_data[Hash.length];
} Finished;
```

That's it! Once you have stripped the handshake header from the message, all that's left is the MAC tag itself. Thus, all you must do in this function is:

1. Derive the `finished_key`. Consider the following from the TLS RFC [1]:
   - `finished_key` = HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)

   Your implementation of `tls_finished_key_derive` will help you here. To save you searching through the TLS RFC (the definition of BaseKey is very well-hidden), BaseKey here is the `remote_hs_traffic_secret`, which you saved into the state in a previous function!
2. Create a transcript hash from the currently maintained *self*.`transcript`. Your previous implementation of `tls_transcript_hash` will help you here.
3. Verify the MAC tag. Your previous implementation of `tls_finished_mac_verify` will help you here.
4. If the signature verifies correctly, update *self*.`transcript` with the CertificateVerify message you just processed.

5. If *self*.`role` is client, then you will also need to compute *self*.`local_ap_traffic_secret`, *self*.`remote_ap_traffic_secret`. You will need to compute a new transcript hash for this step, since you just updated the *self*.`transcript`. Your previous implementation of `tls_derive_secret` will help you here.

## *Testing and Evaluation*

We have provided in the skeleton file two test modules – all of which involve file reads and writes. You can use these modules to test your implementation. The test modules are summarized as follows:

1. The first module tests the correctness of the `tls_handshake.py` functions over various test vectors.
2. The second module tests the correctness of `tls_psk_functions.py` functions over various test vectors.

For each of these test modules, you are provided with the input test vectors and the corresponding output vectors in separate input and output files. When you execute each test module, you will generate an output file. Your implementation is correct if the contents of this file *exactly* match the contents of the output file provided to you. - running the test modules will tell you which functions output correctly.

To run the test modules, simply run **python3 -m unittest** in the terminal, and the unit tests will print the results to the terminal.

**Note:** You are free to test your code using your own custom-designed test modules. However, we will be using the *same* test modules as in the skeleton file to evaluate your submissions under an automated evaluation framework. *Please do not tamper with the test modules in any way to avoid interfering with our automated evaluation frameworks.*

## *Evaluation*

When we evaluate your submissions, we will run the exact same test modules, albeit with respect to our own privately generated input and output files, which will not be made public prior to evaluation. However, for this module we only require that you submit completed versions of `tls_handshake.py` and `tls_psk_functions.py` - you may modify any of the other files in the folder as you please. However, as a result of modifying files you may no longer get accurate evaluations of your own files, so we don't recommend this.

**Summary of Evaluation Criteria.** To summarize, you will be evaluated based on the correctness of your implementation of the individual functions:

1. `tls_13_client_hello` (15 points)
2. `tls_13_process_server_hello` (15 points)
3. `tls_process_server_cert_verify` (10 points)
4. `tls_process_finished` (10 points)

So a total of 50 points is available for this portion of this week's lab.

### *Submission Format*

Your completed submission for week 5 should consist of a *pair* of Python files, and should be named "`tls_handshake.py`" and "`tls_psk_functions.py`" respectively.

You are expected to upload your submission to Moodle. The submission for week 5 should be bundled into a single archive file named "module_2_submission_[insert LegiNo].zip".

In conclusion, *happy coding*!

## *References*


1. Eric Rescorla (2018) "RFC8446: The Transport Layer Security (TLS) Protocol Version 1.3" `https://tools.ietf.org/html/rfc8446`.

2. PyCryptodome. "Crypto.Hash Package" `https://pycryptodome.readthedocs.io/en/latest/src/hash/hash.html`

3. PyCryptodome. "Digital Signature Algorithm (DSA and ECDSA)" `https://pycryptodome.readthedocs.io/en/latest/src/signature/dsa.html`

4. PyCryptodome. "PKCS#1 v1.5 (RSA)" `https://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_v1_5.html`

5. PyCryptodome. "Modern modes of operation for symmetric block ciphers" `https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html`

6. PyCryptodome. "ChaCha20 and XChaCha20" `https://pycryptodome.readthedocs.io/en/latest/src/cipher/chacha20.html`

7. Alex Moneger (2015) "tinyec 0.3.1" `https://pypi.org/project/tinyec/`


## *Appendix: Additional Listings*

Listing 6: TinyEC and Elliptic Curve Cryptography

```python
from tinyec import registry
from Crypto.Cipher import AES
from Crypto.Protocol.KDF import HKDF
from Crypto.Hash import HMAC, SHA256, SHA512
from Crypto.Random import get_random_bytes
import math
import secrets

def compress(pubKey):
        return hex(pubKey.x) + hex(pubKey.y % 2)[2:]

def ec_setup(curve_name):
        curve = registry.get_curve(curve_name)
        return curve

def ec_key_gen(curve):
        sec_key = secrets.randbelow(curve.field.n)
        pub_key = sec_key * curve.g
        return (sec_key, pub_key)

def ec_dh(sec_key, pub_key):
        shared_key = sec_key * pub_key
        return shared_key
```

As you can see, a lot of the details of the underlying elliptic curve operations are hidden at this level. But you can still glean a high-level understanding of how elliptic-curve Diffie-Hellman key-exchange works, and compare it with the traditional variant of Diffie-Hellman key-exchange.

For key generation, an integer $d$ is randomly sampled from $\mathbb{Z}_n$, where $n$ (seen in Listing 1 as curve.field.n) is order of the point $g$ (the point $g$ generates the group of points on the elliptic curve). The integer $d$ serves as the secret key `sk` in `ec_key_gen`. Then, the point $g$ is added to itself $d$ times to compute the public key `pk` in `ec_key_gen`. Computing an ECDH shared secret is again scalar multiplication and can be interpreted simply as adding the public-key `pk` to itself `sk`-many times.