Module 02: Key Exchange and the TLS 1.3 Protocol

Week-05: Implementing the TLS 1.3 Protocol PSK Functions

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 aspects of TLS 1.3. Specifically, we will be implementing functions for the Session Resumption mechanisms used in the TLS 1.3 Protocol.

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 λ, 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 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.

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

  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 λ and outputs a symmetric key k.
- $\mathsf{ctxt} \leftarrow \mathsf{AEAD.Enc}(k, N, ad, \mathsf{msg})$: denotes the encryption algorithm of AEAD. It takes as input a symmetric key k, a nonce N, 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 `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 HKDF library to implement key derivation [7]. You should look through `tls_crypto.py` for more information on how to use the various TLS-specific cryptographic functions that you will need to use.

Overview

In this lab, you will be implementing a series of TLS PSK functions. 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_crypto.py`: This file will allow you to run unit tests and see how well your implementation of the tls-specific cryptographic primitives match ours

- `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 that will help you implement your solutions.
- `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`.

Since this portion of the lab is implementing the TLS 1.3 PSK-based functions, let us take a closer look at `tls_psk_functions.py`, and describe the expected API and operations for each.

Server NewSessionTicket

Listing 1: Server NewSessionTicket

```
def tls_13_server_new_session_ticket(self, server_static_enc_key,
    resumption_secret, csuite):
        raise NotImplementedError
```

In this task, you will be implementing a function that creates a post-handshake message known as the `NewSessionTicket` message, described in Section 4.6.1 of the TLS RFC. According to the RFC, the `NewSessionTicket` message has the following structure:

```
struct {
    uint32 ticket_lifetime;
    uint32 ticket_age_add;
    opaque ticket_nonce<0..255>;
    opaque ticket<1..2^16-1>;
    Extension extensions<0..2^16-2>;
} NewSessionTicket;
```

You should set these values according to the following instructions:

- `ticket_lifetime` is the validity period of the use of the PSK. Set this value to the maximum lifetime of tickets as instructed in the specification, which is 604800 seconds, and given in seconds.
- `ticket_age_add` is a randomly generated value, used to obscure the actual age of preshared keys when sent in `PreSharedKeyExtensions`.
- `ticket_nonce` is a randomly generated value, that is used to generate the PSK. We mandate 8 bytes for the `ticket_nonce`.
- `ticket` is an encrypted PSK, along with enough information for the server to later verify that a PSK is still valid. The specification states that "[the ticket] MAY be either a database lookup key or a self-encrypted and self-authenticated value." Our implementation will take the latter approach. We discuss how the `ticket` value is generated below.
- `extensions` is a single extension: `Early Data Indication`, which allows the server to specify the maximum number of bytes the client will be allowed to send in a 0RTT connection. In our implementation, we have chosen 2^{12} as the maximum number of bytes that the server will allow the client to send. **Note**: We differ from the specification when constructing our `EarlyDataIndicationExtension`. Specifically, we only encode `ExtensionType` in a single byte, and we do not add a length-encoding field for `extension_data` as `max_early_data_size` is a constant size.

```
struct {
uint8 ExtensionType
    select (Handshake.msg_type) {
        case new_session_ticket:   uint32 max_early_data_size;
        case client_hello:         Empty;
        case encrypted_extensions: Empty;
    };
} EarlyDataIndicationExtension;
```

where `uint32 max_early_data_size` is a 4-byte representation of the maximum number of bytes that the client is allowed to send in network (big-endian) order.

The ticket allows the server to recover the PSK from the ticket, when the client returns the ticket in a `PreSharedKeyExtension`. We will be using a common technique that allows for stateless servers to recover necessary information: Session Ticket Encryption. We specify here that the server will always use ChaCha20_Poly1305 to encrypt their tickets. This is an AEAD scheme, so recall that AEAD.Enc$(k, N, ad, ptxt)$ takes four inputs: a key k, which is given as input to the `tls_13_server_new_session_ticket` function as `server_static_enc_key`, a nonce N that will be randomly generated by your implementation, associated data ad (which we will not be using for the encrypted ticket, so there is no need to update the cipher with an ad), and a plaintext $ptxt$. We will specify an 8-byte random nonce for our AEAD-encrypted ticket. We need to include enough information for the server to recover the PSK, and verify that the age of the PSK is not greater than the `ticket_lifetime`, so we specify that the plaintext should be:

$$ptxt = PSK \| \texttt{ticket\_add\_age} \| \texttt{ticket\_lifetime} \| \texttt{csuite}$$

You can assume that `csuite` is stored in the state (i.e. in *self*.`csuite`). To compute the PSK from the `ticket_nonce` and the resumption secret, consider the following from Section 4.6.1 of the TLS RFC:

"The PSK associated with the ticket is computed as:

```
HKDF-Expand-Label(resumption_master_secret, "resumption", ticket_nonce,
Hash.length)
```

You may wish to use the TLS-specific cryptographic primitives given in `tls_crypto.py`.

The addition of the `csuite` allows the server to check that the PSK was derived with a consistent hash function - a requirement of the specification. Encrypt the plaintext, and construct the ticket as the concatenation of the nonce N, the ciphertext $ctxt$ and the output tag τ. In addition, `NewSessionTicket` is considered a handshake message, which means adding a handshake header similarly to `ClientHello` or other handshake messages. We have provided a function `attach_handshake_header` to assist you with this. Afterwards, construct the `NewSessionTicket` message as indicated above. Your function should return the `NewSessionTicket` message in byte format.

Client Parse NewSessionTicket

Listing 2: Client Parse NewSessionTicket

```
def tls_13_client_parse_new_session_ticket(self, resumption_secret, nst_msg):
raise NotImplementedError
```

This function should take as input a `resumption_secret`, and a `NewSessionTicket` message, both in byte-format. The function should parse the `NewSessionTicket` message, derive a PSK, an early secret, and a binder key, according to the key schedule, given in the Appendices.

It should construct a PSK dictionary object which, given a key X should return a value Y according to the table below. Parsing the `NewSessionTicket` message and deriving the secrets according to the key schedule should allow you to compute all values stated below.

Key X	Value Y
“PSK"	psk
“lifetime"	`ticket_lifetime`
“lifetime_add"	`ticket_add_age`
“ticket”	`ticket`
“max_data"	`max_data`
“binder key"	`binder_key`
“csuite"	*self*.`csuite`

The function should return the PSK dictionary object. You can assume that the *self* state has already been initialised with the appropriate ciphersuite i.e. *self*.`csuite` already has a valid ciphersuite.

Client Prepare PskModeExtension

Listing 3: Client Prepare PskModeExtension

```
def tls_13_client_prep_psk_mode_extension(self, modes):
    raise NotImplementedError
```

In this function you will create a PSK extension, which indicates to the server which PSK modes the client supports the use of. The function will take a tuple of integers as input, indicating the PSK modes that it supports, as described in Section 4.2.9:

```
enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

struct {
    PskKeyExchangeMode ke_modes<1..255>;
} PskKeyExchangeModes;
```

Correction: Please note that we diverge from the specification when creating the `PreSharedKeyExchangeModes` in the following way: the PskKeyExchangeModes extension is constructed without the length encoding for extension data in the following way: 2 bytes for Extension Type, followed by the PskKeyExchangeModes struct as defined in the RFC (variable length).

Finally, the function should output a valid `PskKeyExchangeModes` extension in byte format.

Client Prepare PSK Extension

Listing 4: Client Prepare PSK Extension

```
def tls_13_client_prep_psk_extension(self, PSKS, ticket_age, transcript):
        raise NotImplementedError
```

In this function, you will create a PSK extension, which will indicate to the server the list of `OfferedPsks` and identities that the client is willing to support. The extension will take as input: `PSKS`, a tuple of PSK dictionaries, like the one you created in `tls_13_client_parse_new_session_ticket`. `ticket_age`, a tuple of integers corresponding to the ages of each PSK in `PSKS` in milliseconds. You can assume that `ticket_age` is order in the same way as `PSKS`, and has the same length. Finally, `transcript`, a representation of the `ClientHello` message that already has the length-encoding fields set appropriately. This detail will become clearer later. Your implementation should use all PSK dictionaries in `PSKS` to generate a `PreSharedKeyExtension` according to the specification described in Section 4.2.11. The `extension_data` field of this extension contains a `PreSharedKeyExtension`. See the following details below:

```
struct {
    opaque identity<1..2^16-1>;
```

```
    uint32 obfuscated_ticket_age;
} PskIdentity;


opaque PskBinderEntry<32..255>;

struct {
    PskIdentity identities<7..2^16-1>;
    PskBinderEntry binders<33..2^16-1>;
} OfferedPsks;


struct {
    select (Handshake.msg_type) {
        case client_hello: OfferedPsks;
        case server_hello: uint16 selected_identity;
    };
} PreSharedKeyExtension;


struct {
    uint16 ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;
```

It will be useful to note that the`PreSharedKeyExtension` is the input `extension_data` here. Note here that `identity<1..`2^{16}`-1>` corresponds to the `ticket` value saved in PSK dictionary, and `obfuscated_ticket_age` is a representation of the ticket age (recall that the age of the ticket is indexed identically to the index of the corresponding PSK dictionary) added to the "`lifetime_add`" value saved in the PSK dictionary, modulo 2^{32}. This is described in Section 4.2.11.1 of the TLS RFC. If the `ticket_age` of the PSK is greater than `ticket_lifetime`, the implementation should not include this PSK.

The binder value forms a binding between the PSK, the current handshake, and the handshake from which it was generated. Each entry in the binders list is computed as a HMAC over the transcript hash, containing a `ClientHello` message and including the `PreSharedKeyExtension.identities`. The length fields for the messages (including the overall length, the length of the extension block, and the length of the PSK extension) are set as if the binders of the correct lengths are present. As we specified earlier, we assume that the "length" fields for the `ClientHello` have already been set appropriately, so you only need to worry about the extension length field for the `PreSharedKeyExtension` itself.

Thus, you will need to compute the identities list before you can compute the binders themselves, and then determine what is the expected full length of the extension. Each PSK dictionary contains the integer representation of the `csuite` you will compute each binder with, and that, paired with `hash.digest_size`(), will allow you to precompute the expected lengths of the binders list. Don't forget to include the length encoding of the binders list into your expected extension length.

Correction: Please note that we diverge from the specification for computing the binder values in the following way: For Binder value computations, BinderKey is passed directly to the `tls_finished_mac` instead of computing a key through the `tls_finished_key_derive` function. Thus `PskBinderEntry` is computed as the `Finished` message, but with the `FinishedKey` instead being the `binder_key` included in the PSK dictionary. Once you have the list of binder values, create the final `PreSharedKeyExtension`, and return the `PreSharedKeyExtension` in byte format.

Server Prepare PSK Extension

Listing 5: Server Parse PSK Extension

```
def tls_13_server_parse_psk_extension(self, server_static_enc_key, psk_extension,
    transcript)
        raise NotImplementedError
```

In this function, you will parse the `PreSharedKeyExtension`, and iteratively use the `server_static_enc_key` to sequentially decrypt `tickets`, and recover the PSK, `ticket_add_age` and `ticket_lifetime` values. Then, you will use the PSK to generate a binder key. You must verify that the `csuite` indicated in the `ticket` matches the server's negotiated ciphersuite (contained in *self*.`csuite`). Finally, that the binder value verifies, with the given `ClientHello` (given as an opaque value in the transcript), and the given PSK extension. Remember to truncate the PSK extension before you try to verify!

The function should return the first such PSK and the index of the selected identity, where those conditions are met. Otherwise, the server should move onto the next `identity` and `binder` value in the extension. If no such `identity/binder` pairs are valid, the server should return `DecryptError`.

In the test vectors we give, at least on of the `identity/binder` pairs should validate correctly, given the conditions outlined above.

TLS Key Schedule

Listing 6: TLS Key Schedule

```
def tls_13_psk_key_schedule(self, psk_secret, dhe_secret, transcript_one,
    transcript_two, transcript_three, transcript_four)
        raise NotImplementedError
```

This last function is meant to evaluate how well you can follow the TLS key schedule, and how to derive the various secrets given in `tls_crypto.py`.

Simply put, the function takes `psk_secret`, `dhe_secret` as the PSK and DHE inputs into the key-schedule, derive and then return the following secrets (in order!) using the `tls_crypto.py` functions:

1. `early_secret`,
2. `binder_key`,
3. `client_early_traffic_secret`,
4. `client_early_key`,
5. `client_early_iv`,
6. `early_exported_master_secret`,
7. `derived_early_secret`,
8. `handshake_secret`,
9. `client_handshake_traffic_secret`,
10. `client_handshake_key`,
11. `client_handshake_iv`,
12. `server_handshake_traffic_secret`,

13. `server_handshake_key`,
14. `server_handshake_iv`,
15. `derived_handshake_secret`,
16. `master_secret`,
17. `client_application_traffic_secret`,
18. `client_application_key`,
19. `client_application_iv`,
20. `server_application_traffic_secret`,
21. `server_application_key`,
22. `server_application_iv`,
23. `exporter_master_secret`,
24. `resumption_master_secret`

You can assume that:

- `transcript_one` = `ClientHello`
- `transcript_two` = `ClientHello..ServerHello`
- `transcript_three` = `ClientHello..ServerFinished`
- `transcript_four` = `ClientHello..ClientFinished`

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_server_new_session_ticket` (10 points)
2. `tls_13_client_parse_new_session_ticket` (7.5 points)
3. `tls_13_client_prep_psk_mode_extension` (2.5 points)
4. `tls_13_client_prep_psk_extension` (15 points)
5. `tls_13_server_parse_psk_extension` (10 points)
6. `tls_13_psk_key_schedule` (5 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/

Key Schedule

(On Next Page)

```
            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
```