Module 02: Key Exchange and the TLS 1.3 Protocol

Week-04, Part II: Implementing the TLS 1.3 Protocol

Jan Gilcher, Fernando Virdia, Kenny Paterson, and Lukas Burkhalter

jan.gilcher@inf.ethz.ch, fernando.virdia@inf.ethz.ch
kenny.paterson@inf.ethz.ch, lubu@inf.ethz.ch

October 2021

Welcome to the second lab for this module. This lab is part of the Module-02 on *Key Exchange and TLS 1.3*. Due to technical limitations the labs will only take place in person. 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/mod/forum/view.php?id=616927`

Students who cannot attend the labs in person are especially encouraged to use Moodle to ask questions.

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 cryptographic primitives you have seen in previous parts, 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 (>=3.4) 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 APIs, 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 for part I, for this lab you will need to install an elliptic-curve library using `pip3 install tinyec`, and modern AEAD libraries using PyCryptodome (`pip3 install pycryptodomex`). 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 favourite 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 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 in place of scalar multiplication $[x]P$ where P is a base 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 **Cryptodome.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 **Cryptodome.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 algorithm of a digital signature scheme SIG. It takes as input a security parameter λ, and outputs a signing key sk and a verification key vk.
- $\sigma \leftarrow_R \mathsf{SIG.Sign}(\mathsf{sk}, \mathsf{msg})$: denotes the signing algorithm of SIG. It takes as input a signing key sk and a message msg, and outputs a signature σ.
- $\{0, 1\} \leftarrow \mathsf{SIG.Verify}(\mathsf{vk}, \mathsf{msg}, \sigma)$: denotes the verification algorithm of SIG. It takes as input a verifying key vk, a message msg, and a signature σ 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 **Cryptodome.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 msg, and outputs a MAC tag τ.

  In this lab, we will be using HMAC to instantiate our MAC algorithm, imported from **Cryptodome.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.
- $\mathsf{ctxt} \leftarrow \mathsf{AEAD.Enc}(k, \mathit{nonce}, \mathit{ad}, \mathsf{msg})$: 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
- $\{\mathsf{ctxt}, \perp\} \leftarrow_R \mathsf{AEAD.Dec}(k, \mathit{nonce}, \mathit{ad}, \mathsf{ctxt})$: denotes the decryption 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:

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

  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 **Cryptodome.Cipher** [4].

Implemented Functions

Before we begin, it is worth highlighting some code that has already been provided to you in the skeleton file `tls_crypto.py`. 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 them 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 handled mostly transparently to the user.

In what follows, we will be focusing on implementing client-specific functions that are called during the Handshake, 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
from tls_application import TLSConnection

def client_socket():
  s = socket.socket()
  host = socket.gethostname()
  port = 1189
  s.connect((host, port))
  client = TLSConnection(s)
  client.connect()
  client.write("challenge".encode())
  msg = client.read()
  print(msg.decode('utf-8'))
  s.close()

if __name__ == '__main__':
  client_socket()
```

This is a fairly simple function, and we can go through how it works. The `client_socket` function begins by creating a socket object, allowing network communication. Afterwards, the `simple_client` function connects to the (*host*, *port*) pair provided and intialises a TLSConnection. TLSConnection uses a similar, but much simplified, as openssl's connection interface, i.e. the client calls `connect()` to initiate the handshake.

After a successful handshake the client can then simply send and read messages via TLSConnection's `write()` and `read()` calls.

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.

TLS Cryptographic Functions

In part II of this week's lab, you will be implementing a series of cryptographic primitives used in TLS. 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. **Most of these files will be completed in next week's lab.** In this weeks lab, all your changes will be in `tls_crypto.py`, while the next weeks TLS parts II and III will build upon the cryptographic functions you implemented this week to implement the Handshake protocol and then add support for TLS session resumption and 0-RTT. At the end of this lab all your changes will be in the `tls_crypto.py`, which together with `tls_handshake.py`, `tls_psk_handshake.py`, and `tls_psk_state_machines.py` from next week should include all your work on TLS.

- `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 file 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 matches 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 matches 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_layer, respectively.
- `tls_crypto`: This file contains the tls-specific functions that you will be implementing for this assessment.
- `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 preparation and negotiation of TLS extensions sent during the ClientHello and ServerHello messages.
- `tls_handshake`: This file contains code for the handshake for both client and server handshake functions, some of which you will implement next week.
- `tls_psk_handshake`: This file contains and extended version of the Handshake with added PSK functionality that you will implement for both clients and servers next week.
- `tls_record_layer`: This file contains high-level API for preparing both plaintext and encrypted TLS record packets.
- `tls_state_machines`: This file contains the simplified TLS Handshake state machines.

- `tls_psk_state_machines`: This file contains the simplified TLS Handshake state machines. You will be extending the state machines in this file to support session resumption and 0-RTT data next week.
- `psk_client.py`: This file creates sockets and manages networking tasks for a client TLS instance. After a succesful Handshake it tries to resume the session using PSKs and sends early data.
- `psk_server.py`: This file creates sockets and manages networking tasks for a server TLS instance. It supports PSK and early data.

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 require 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 be 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 require 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`.

Lab Task

Your task for this lab is to implement a series of 14 different cryptographic functions that will be used in our TLS implementation (that you will complete in next week's lab). Skeletons for each of the unimplemented functions can be found in `tls_crypto.py`. We consider each of them in turn, describing the expected API and operations for each. See listings 2–15 that follow.

Listing 2: TLS Transcript Hash

```
def tls_transcript_hash(csuite, context):
        raise NotImplementedError
```

This function is straightforward, but essential to TLS – computing a hash digest over the transcript of messages sent between the two communicating parties. The `tls_transcript_hash` function will take as input `csuite` – an integer representing a negotiated ciphersuite, and `context` – a series of bytes. The output of this function should be `transcript_hash`: a hash digest computed over `context`, also a series of bytes.

The file `tls_constants.py` contains the definitions of `csuite` integers used both internally in the TLS implementation, and externally when communicating with the server. You may wish to examine the PyCryptodome documentation on the use of **Cryptodome.Hash** functions [2].

Remember: your implementation is required to support a number of different ciphersuites. The hash function used to compute the digest will depend on which ciphersuite is indicated via `csuite`.

Listing 3: TLS HKDF Label

```
def tls_hkdf_label(label, context, length):
        raise NotImplementedError
```

Whenever TLS calls HKDF.Expand for the generation of symmetric keys, it specifies a particular mechanism for computing the "info" input. The function `tls_hkdf_label` is our implementation of this mechanism. Consider the following from Page 91 of the TLS 1.3 RFC [1]:

> The key derivation process makes use of the HKDF-Extract and HKDF-Expand functions as defined for HKDF, as well as the functions defined below:
>
> ```
> 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;
> ```

All values, here and elsewhere in the specification, are transmitted in network byte (big-endian) order; so uint16 is a 2-byte representation of Length in network byte (big-endian) order. Since `label` and `context` are both variable-length vectors, they are preceded by a length-field. The length-field will be in the form of a N-byte representation of the length, where N is the number of bytes required to hold the maximum length. The function tls_hkdf_label takes as input `label` and `context`, which will be given as a series of bytes, and `length`, which will be an integer value. The function will output `hkdf_label` as a series of bytes. We have provided an implementation of HKDF within `tls_crypto.py`, which you should examine.

Listing 4: TLS Derive Key IV

```
def tls_derive_key_iv(csuite, secret):
        Raise NotImplementedError
```

This function is straightforward: compute the TLS traffic keying material, specifically the key and IV. The function takes as input `csuite` –an integer representation of the negotiated ciphersuite (which specifies the underlying cryptographic primitves used to build the TLS-specific cryptographic primitives), and `secret` – a series of secret bytes. The function should output (`key`, `iv`), both returned as a series of bytes. The following from the TLS 1.3 RFC [1] might be helpful for you:

> The traffic keying material is generated from the following input values:
>
> - A secret value
>
> - A purpose value indicating the specific value being generated
>
> - The length of the key being generated
>
> The traffic keying material is generated from an input traffic secret value using:

```
[sender]_write_key = HKDF-Expand-Label(Secret, "key", "", key_length)
[sender]_write_iv  = HKDF-Expand-Label(Secret, "iv", "", iv_length)
```

Listing 5: TLS Extract Secret

```
def tls_extract_secret(csuite, keying_material, salt):
        raise NotImplementedError
```

This function is similarly straightforward: It will initialise an HKDF object and uses the underlying Extract function to extract a secret from the keying material and salt inputs given. The function takes as input `csuite` – an integer representation of the negotiated ciphersuite (which specifies the underlying cryptographic primitves used to build the TLS-specific cryptographic primitives), `keying material` – a series of secret (potentially non-uniform) bytes, and `salt` – a series of bytes. The function should output `secret` as a series of bytes.

Listing 6: TLS Derive Secret

```
def tls_derive_secret(csuite, secret, label, messages):
        Raise NotImplementedError
```

Now we begin to see some usage of our previously-implemented functions. Consider the following from the TLS 1.3 RFC [1]:

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

As a result, you will need to use `tls_transcript_hash` to implement this function. In addition, you might recall that we also saw the definition of HKDF-Expand-Label above. This function takes as input: `csuite` – an integer representation of the negotiated ciphersuite (which specifies the underlying cryptographic primitves used to build the TLS-specific cryptographic primitives); `secret` – a series of secret uniform bytes; `label` – a series of bytes; and `messages` – a series of bytes. The function should output `secret` as a series of bytes.

Listing 7: TLS Finished Key Derive

```
def tls_finished_key_derive(csuite, secret):
        NotImplementedError
```

This is another function where we get to see some usage of our previously-implemented functions. This function takes as input: `csuite` – an integer representation of the negotiated ciphersuite (which specifies the underlying cryptographic primitves used to build the TLS-specific cryptographic primitives) and `secret` – a series of secret uniform bytes, which is the BaseKey input referred to below. The function should output `finished_key` as a series of bytes. Consider the following from the TLS 1.3 RFC [1]:

```
finished_key =
HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)
```

Listing 8: TLS Finished MAC

```
def tls_finished_mac(csuite, key, context):
        NotImplementedError
```

We stay on the theme of previously-implemented functions. This function takes as input: `csuite` – an integer representation of the negotiated ciphersuite (which specifies the underlying cryptographic primitves used to build the TLS-specific cryptographic primitives), `key` – the finished key referenced earlier, given as a series of secret uniform bytes, and `context` – the transcript hash, given as a series of bytes. The function should output `finished_tag`, the output of a MAC algorithm, as a series of bytes. Consider the following from the TLS 1.3 RFC [1]:

> The verify_data value is computed as follows:
>
> ```
> verify_data =
>     HMAC(finished_key,
>         Transcript-Hash(Handshake Context,
>             Certificate*, CertificateVerify*))
> ```
>
> * Only included if present.

Our implementation only has Server authentication, and as a result Certificate and CertificateVerify will not be included. You may wish to consult the PyCryptodome documentation of the usage of HMAC [2]. Remember: your implementation is required to support a number of different ciphersuites. The underlying hash function used to compute the digest will depend on which ciphersuite is indicated via `csuite`.

Listing 9: TLS Finished MAC Verify

```
def tls_finished_mac_verify(csuite, key, context, tag):
        NotImplementedError
```

The function `tls_finished_mac_verify` is similar to the previously defined function, verifying a MAC tag instead of generating a MAC tag. This function takes as input:

`csuite` – an integer representation of the negotiated ciphersuite (which specifies the underlying cryptographic primitves used to build the TLS-specific cryptographic primitives), `key` – the finished key referenced earlier, given as a series of secret uniform bytes, `context` – the transcript hash, given as a series of bytes and `tag`, a HMAC tag given as a series of bytes. The function should verify `tag`, and not return any output. As above, you may wish to consider the PyCryptodome documentation of the use of HMAC [2].

Listing 10: TLS Nonce

```
def tls_nonce(csuite, sqn_no, iv):
        NotImplementedError
```

This function will construct a TLS-specific nonce given a previously-generated IV and a sequence number. This function takes as input: `csuite` – an integer representation of the negotiated ciphersuite (which specifies the underlying cryptographic primitves used to build the TLS-specific cryptographic primitives), `sqn_no` – an integer that counts the number of encryptions per key, and `iv` – the IV generated earlier via `tls_derive_key_iv`, given as a series of bytes. The function should output `nonce`, a series of bytes. To construct the nonce, consider the following from the TLS 1.3 RFC:

> The per-record nonce for the AEAD construction is formed as follows:
>
> 1. The 64-bit record sequence number is encoded in network byte order and padded to the left with zeros to iv_length.
>
> 2. The padded sequence number is XORed with either the static client_write_iv or server_write_iv (depending on the role).
>
> The resulting quantity (of length iv_length) is used as the per-record nonce.

In `tls_crypto.py` we have given a function `xor_bytes` which XORs two byte strings of identical lengths, which you can use to perform the XOR function here. In addition, in `tls_constants.py`, we define a dictionary `IV_LEN` which, given a `csuite` key, will return the IV length (as an integer) associated with that ciphersuite.

Listing 11: TLS AEAD Encrypt

```
def tls_aead_encrypt(csuite, key, nonce, plaintext):
        NotImplementedError
```

In this function, you will perform the AEAD encryption of `plaintext`, given the `key` and `nonce`. This function takes as input: `csuite` – an integer representation of the negotiated ciphersuite (which specifies the underlying cryptographic primitves used to build the TLS-specific cryptographic primitives), `key` – the AEAD key that will be used to encrypt the plaintext, given as a series of bytes, `nonce` – the TLS-specific nonce generated earlier via `tls_nonce`, given as a series of bytes, and `plaintext`, given as series of bytes. The function should output `ciphertext`, a series of bytes. Recall the AEAD encryption function defined earlier: $\mathsf{ctxt} \leftarrow \mathsf{AEAD.Enc}(k, nonce, ad, \mathsf{msg})$. Thus, we also need to consider how to construct the Additional Data (AD) input. To construct AD, consider the following from the TLS 1.3 RFC [1]:

```
additional_data = TLSCiphertext.opaque_type ||
```

```
TLSCiphertext.legacy_record_version ||
TLSCiphertext.length
```

The AD field for TLS 1.3 is the record header: the `opaque_type` refers to the message type: since this is a TLS ciphertext, this is an "Application" type. `legacy_record_version` refers to TLS 1.2. This was chosen to improve interoperability with outdated middlebox devices that potentially drop TLS messages with unknown versions. In `tls_constants.py` we have defined integer representations of both fields. However you will need to convert both to bytes representations of these integers, to give the AD as input to the AEAD algorithm. You may wish to read the PyCryptodome documentation on the use of AEAD schemes [5] as well as the usage of ChaCha20Poly1305 [6]. Remember: your implementation is required to support a number of different ciphersuites. The AEAD scheme used to encrypt the message will depend on which ciphersuite is indicated via `csuite`.

Listing 12: TLS AEAD Decrypt

```
def tls_aead_decrypt(csuite, key, nonce, ciphertext):
        NotImplementedError
```

We now turn to the AEAD decryption algorithm, which will work very similarly to the AEAD encryption algorithm. This function takes as input: `csuite` – an integer representation of the negotiated ciphersuite (which specifies the underlying cryptographic primitves used to build the TLS-specific cryptographic primitives), `key` – the AEAD key that will be used to encrypt the plaintext, given as a series of bytes, `nonce` – the TLS-specific nonce generated earlier via `tls_nonce`, given as a series of bytes, and `ciphertext`, given as series of bytes. The function should decrypt (and verify) the `ciphertext` and output `plaintext`, a series of bytes. As before, you will have to construct the Additional Data (AD) input yourself. As before, you may wish to read the PyCryptodome documentation on the use of AEAD schemes [5] as well as the usage of ChaCha20Poly1305 [6].

Listing 13: TLS Signature Context

```
def tls_signature_context(context_flag, content):
        NotImplementedError
```

This function will transform `content` into a TLS-conforming message ready to be signed. The function takes as input `context flag` – a string that is either equal to "SERVER" or "CLIENT", depending on which party is signing the TLS-conforming message, and `content` – the content to be signed. The function will return `message`, a series of bytes. In `tls_constants.py` we give definitions for both `SERVER_FLAG` and `CLIENT_FLAG` to make implementing this function easier. To transform this content, consider the following from the TLS 1.3 RFC [1]:

> The digital signature is then computed over the concatenation of:
>
> - A string that consists of octet 32 (0x20) repeated 64 times
>
> - The context string
>
> - A single 0 byte which serves as the separator

- The content to be signed

The context string for a server signature is "TLS 1.3, server CertificateVerify". The context string for a client signature is "TLS 1.3, client CertificateVerify".

For example, if the transcript hash was 32 bytes of 01 (this length would make sense for SHA-256), the content covered by the digital signature for a server CertificateVerify would be:

20202020202020202020202020202020202020202020202020202020202020

20202020202020202020202020202020202020202020202020202020202020

544c5320312e332c207365727665722043657274696669636174655665726966

79

00

0101010101010101010101010101010101010101010101010101010101010101

To be consistent with the rest of our encodings, however, we diverge from the specification and *do not* encode in hex. Instead we encode our context string into bytes as follows: `context_string.encode()`.

Listing 14: TLS Signature Generation

```
def tls_signature(signature_algorithm, msg, context_flag):
        NotImplementedError
```

This function will straightforwardly sign a message `msg`, transform the message to a TLS-conforming message using `tls_signature_context`, and then sign it using `signature_algorithm`. The function takes as input `signature_algorithm` – an integer representation of the negotiated signature algorithm (which specifies the underlying signature and hash primitives used to build the TLS-specific signing algorithm), `msg` – the message to be signed, given as bytes, and `context_flag` – the string indicating whether the server or client is signing the message. To implement this function, you will need to use the appropriate secret key. In `tls_constants`, we have given you access to `RSA2048_KEY` and `SECP384R1_KEY`, the secret key that will be used by the server to sign messages, depending on whether RSA or ECDSA signatures have been negotiated. The function will return `signature`, as a series of bytes. You may wish to read the PyCryptodome documentation on the use of ECDSA [3] and RSA signatures [4]. Remember: your implementation is required to support a number of different signature schemes. The signature scheme used to encrypt the message will depend on which algorithm is indicated via `signature_algorithm`.

Listing 15: TLS Signature Verify

```
def tls_verify_signature(signature_algorithm, message, context_flag, signature,
    public_key):
        NotImplementedError
```

This is the final TLS cryptographic primitive to implement for this lab. Straightforwardly, `tls_verify_signature` will verify a given `signature` over a message (that you will be required to use `tls_signature_context` to transform to an TLS-conforming message) using the `public_key` associated with `signature_algorithm`. The function

takes as input `signature_algorithm` – an integer representation of the negotiated signature algorithm (which specifies the underlying signature and hash primitives used to build the TLS-specific signing algorithm), `message` – the message to be signed, given as bytes, `context_flag` – the string indicating whether the server or client is signing the message, `signature` – the signature to be verified, given as a series of bytes, and `public_key`, the public key to use to verify the signature, given as either an RSA or DSS public key. The function will verify the `message`, and not return an output. As before, you may wish to read the PyCryptodome documentation on the use of ECDSA [3] and RSA signatures [4].

Testing

We have provided a python unit test file `test_tls_crypto.py` to test your implementation. The unit test class tests the correctness of the `tls_crypto.py` functions over various test vectors. For this test module, you are provided with the input test vectors and the corresponding output vectors in separate input and output files. When you execute the test module, you will generate an output file. Your implementation is likely correct if the contents of this file *exactly* match the contents of the output file provided to you. Running the test module will tell you which functions output correctly.

Note: It might be a good idea to test your code using your own custom-designed test modules. However, we will be using test modules like the ones in the skeleton file to evaluate your submissions under an automated evaluation framework.

Evaluation

When we evaluate your submissions, we will run similar tests, however, we use private input vectors which can contain edge cases that were not covered by the tests we gave you. These will not be made public prior to evaluation. For this module we require that you submit completed versions of "`tls_crypto`". 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_transcript_hash` (1 point)
2. `tls_hkdf_label` (2 points)
3. `tls_derive_key_iv` (2 points)
4. `tls_extract_secret` (1 point)
5. `tls_derive_secret` (2 points)

6. `tls_finished_key_derive` (2 points)
7. `tls_finished_mac` (2 points)
8. `tls_finished_mac_verify` (2 points)
9. `tls_nonce` (2 points)
10. `tls_aead_encrypt` (3 points)
11. `tls_aead_decrypt` (3 points)
12. `tls_signature_context` (3 points)
13. `tls_signature` (2 points)
14. `tls_verify_signature` (3 points)

So a total of 30 points is available for part II of this week's lab.

Submission Format

Your completed submission for part II of this week's lab should consist of a *single* Python file, and should be named "`tls_crypto.py`".

You are expected to upload your submission to Moodle. The submissions for the full module (i.e. weeks 4 and 5) should be bundled into a single archive file named "module_2_submission_[insert LegiNo].zip". In summary, this file should contain the following files:

- "`sigma.py`", your implementation of the SIGMA protocol from the week 4 lab part I.
- "`tls_crypto.py`", your implementation of the cryptographic functions used by TLS 1.3 from the week 4 lab part II.
- "`tls_handshake.py`", your implementation of the TLS 1.3 Handshake from the week 5 lab.
- "`tls_psk_handshake.py`", your implementation of the PSK functionality of TLS 1.3 from the week 5 lab.
- "`tls_psk_state_machines.py`", your implementation of the TLS 1.3 state machine with PSK and 0-RTT support from the week 5 lab.

6 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. "Cryptodome.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 16: TinyEC and Elliptic Curve Cryptography

```
from tinyec import registry
from Cryptodome.Cipher import AES
from Cryptodome.Protocol.KDF import HKDF
from Cryptodome.Hash import HMAC, SHA256, SHA512
from Cryptodome.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 gain 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.