Assignment 1: Elliptic curve Diffie-Hellman (X25519)

Muhammad Ayain Fida Rana
Department of Computer Science and Technology
University of Cambridge
15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom
mafr2@cam.ac.uk

1 Introduction

In this assignment, I have implemented the X25519 Diffie-Hellman key exchange, based on the Curve25519 elliptic curve, from scratch. My implementation supports two scalar multiplication algorithms: 1) Montgomery curve group law with a double-and-add algorithm, and 2) the Montgomery ladder. My implementation closely follows RFC 7748 (Langley et al., 2016) and Martin's tutorial (Kleppmann, 2020). The code is publicly available at https://github.com/ayainfida/p79-assignment1-x25519.

2 Implementation Architecture

The core implementation is split across modular components, briefly introduced here, and details follow in Section 3.

Finite Fields (`field.py`). Provides arithmetic operations (addition, subtraction, multiplication, squaring, inversion, and square-root operations.) on the prime field $\mathbb{F}_p$.

Group Laws (`group_law.py`). Implements point addition and point doubling for Curve25519.

Encoding/Decoding (`encoding.py`). Handles conversion between bytes and integers for both scalars and x-coordinates.

Scalar Multiplication Methods (methods.py). Provides two scalar multiplication algorithms: a montgomery ladder and double-and-add.

X25519 (`api.py`). Exposes a public X25519 interface to generate a public/private key pair and then compute a shared secret.

Defaults (`default.py`). Defines shared constants used throughout the implementation.

3 Implementation Design

3.1 Finite Field Operations

Curve25519 is defined by the elliptic curve (Figure 1):

$$y^2 = x^3 + Ax^2 + x \quad \text{over the field } \mathbb{F}_p,\ p = 2^{255} - 19,\ A = 486662. \tag{1}$$

I did not attempt to justify the choice of parameters apart from their role in RFC 7748 (Langley et al., 2016). Specifically, the size of p determines the difficulty of the discrete logarithm problem and the performance of the curve operations.

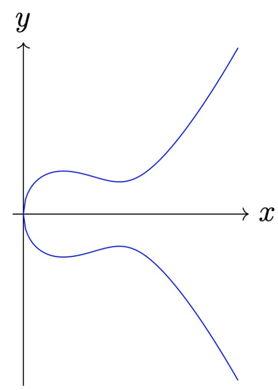


Figure 1: Geometric visualization of an elliptic curve over the real numbers.

I represented the field elements using Python's `int` $\pmod{p}$ after each operation. My implementation supports basic field operations including addition, subtraction, multiplication, squaring, inversion[1], and division[2].

Moreover, the double-and-add algorithm (detailed in Section 3.3.1) also requires the y-coordinate. I could have passed a valid y as described by Martin in class, but to avoid having different X25519 interfaces for both algorithms, I chose to calculate y, which required me to solve the following equation:

$$y = \pm\sqrt{x^3 + Ax^2 + x} \tag{2}$$

To compute the square root, I then defined an `fsqrt` operation based on the exponentiation-based method applicable to primes $p \equiv 5 \pmod{8}$, as described in RFC 8032 (Josefsson & Liusvaara, 2017).

3.2 Representation and Encoding

I defined a `Point` dataclass that stores (x, y) coordinate on Curve25519. If only x is provided, y is computed using the method `calculate_y(x)`. For the case where both (x, y) are provided, it validates this by checking if the following equation holds:

$$y^2 \pmod{p} \equiv x^3 + Ax^2 + x \pmod{p} \tag{3}$$

Moreover, according to the RFC 7748 specification (Langley et al., 2016):

- All `32-byte` scalars and coordinates are encoded/decoded using the little-endian convention.
- The MSB of the last byte must be cleared when decoding the x-coordinate.

```
b = b[:-1] + bytes([b[-1] & 0x7F]) # unsets the MSB
return decode_little_endian(b) % P
```

  The modulo p allows non-canonical values between $2^{255} - 19$ and $2^{255} - 1$ to be valid field elements as well.
- The scalars are clamped: 1) clear the 3 LSBs of the first byte, 2) clear the MSB of the last byte, and 3) set the second MSB of the last byte.

[1] implemented using Fermat's little theorem as in the lecture slides
[2] by multiplying with its inverse

```
k_arr = bytearray(k)
k_arr[0] &= 248
k_arr[31] &= 127
k_arr[31] |= 64
```

3.3 Scalar Multiplication

3.3.1 *Double-and-Add*

This method implements scalar multiplication using group laws. Given a scalar k and a valid point P, the algorithm recursively computes kP by repeated point doubling and point addition based on the value of k.

```
def double_and_add(k: int, Pt: Point) -> Point | PointAtInfinity:
    if k == 1:
        return Pt
    # k is evev: Pt is doubled
    elif k & 1 == 0:
        return point_doubling(double_and_add(k // 2, Pt))
    # k is odd: Pt is doubled and then Pt is added
    else:
    return point_addition(point_doubling(double_and_add((k - 1) // 2, Pt)), Pt)
```

One thing to note here is that this method requires the complete point (x, y). This is because point addition on an elliptic curve is geometrically defined (Figure 2) considering the straight line that passes through the two points[3] that are added.

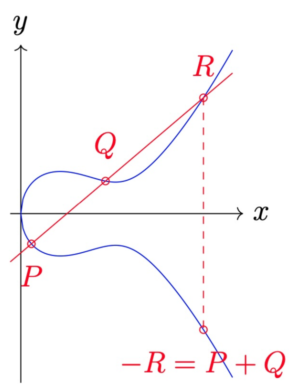


Figure 2: Geometric illustration of elliptic curve point addition over the real numbers.

This line has a slope λ, which depends on both x and y and is used to determine the resulting point (x_3, y_3), as shown in the equations[4] below:

$$x_3 = \lambda^2 - A - x_1 - x_2 \quad (4)$$
$$y_3 = \lambda(x_1 - x_3) - y_1 \quad (5)$$

3.3.2 *Montgomery Ladder*

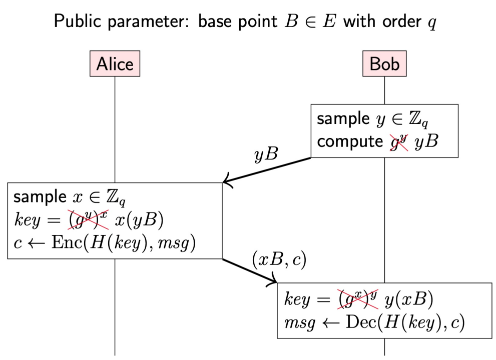
This method implements scalar multiplication using only x-coordinates. I implemented it following the code given in RFC 7748 (Langley et al., 2016) and Martin's tutorial (Kleppmann,

[3] distinct
[4] following from Martin's tutorial (Kleppmann, 2020)

2020). The idea is that it maintains two points that are a step apart and updates them in a fixed[5] pattern, unlike data-dependent sequences in double-and-add. In each iteration, the current scalar bit determines which of the two points represents the correct intermediate result. Moreover, projective coordinates of the form $(X : Z) \equiv X/Z$ are used internally to delay field inversions[6] until the end, and are performed only once: `fdiv(x_2, z_2)`.

3.4 X25519 Key Exchange

The implemented X25519 interface allows setting the scalar multiplication algorithm to either `LADDER` or `DOUBLE_AND_ADD`. Each party generates a private key and derives the corresponding public key. They then compute a shared secret by applying X25519 to their private key and the peer's public key, which can then be hashed, e.g., using `SHA256`, to obtain a symmetric key.




```
# The code snippet demonstrates the relevant API calls.
x25519_instance = X25519(LADDER)

# Alice generates her key pair
alice_sk = x25519_instance.generate_private_key()
alice_pk = x25519_instance.derive_public_key(alice_sk)

# Bob generates his key pair
bob_sk = x25519_instance.generate_private_key()
bob_pk = x25519_instance.derive_public_key(bob_sk)

# Shared secret computation
alice_shared_secret = x25519_instance.x25519(alice_sk, bob_pk)
bob_shared_secret = x25519_instance.x25519(bob_sk, alice_pk)

# Derive a symmetric key
symmetric_key = sha256(alice_shared_secret).digest()
```

Figure 3: Elliptic Curve Diffie-Hellman Key Exchange

4 Testing and Validation

Individual tests are documented in `tests/*.py`; here, I provide a high-level overview of the testing strategy.

[5] 18 arithmetic operations
[6] costly in terms of performance

X25519 API Testing. Validated my implementation against the official test vectors published in RFC 7748 (Langley et al., 2016), covering both single-shot and iterated scalar multiplication tests, and verified correct handling of invalid input lengths.

Scalar Multiplication Agreement. Verified that both scalar multiplication methods produce identical results for the same inputs.

Diffie–Hellman (DH) Key Exchange and Agreement. Tested DH key exchange with independent key pairs for both parties and verified that they derive the same shared secret. The key pairs were sourced from: a) RFC 7748 (Langley et al., 2016), b) pycurve25519 (TomCrypto, 2013), and c) randomly generated private keys.

Encoding. Tested little-endian decoding/encoding, x-coordinate MSB masking (RFC 7748), and scalar clamping.

Finite Field Operations Tested for correctness of the defined operations and verified algebraic properties (commutativity, associativity, identity, inverse) for addition and multiplication.

Group Law Operations Validated the results of point addition and doubling are valid points on curve, along with edge cases like the point at infinity.

5 Discussion

5.1 Limitations and Production Considerations

I used Python's built-in `int` type for field arithmetic over $\mathbb{F}_p$, which is not constant-time. This means that operations on values close to p may take longer than on smaller values, potentially leaking information through timing-based side-channel attacks. Therefore, my Montgomery ladder, despite its fixed pattern of operations, is vulnerable to such attacks. For a production-quality implementation, constant-time field arithmetic, uniform memory accesses, and avoidance of data-dependent branches[7] would be required.

Moreover, the double-and-add scalar multiplication method requires a valid (x, y) coordinate. While this holds for base point multiplication, it may fail for points that do not lie on the curve. I implemented `fsqrt` to compute the y-coordinate, but this is not tested except by following the methodology described in RFC 8032 (Josefsson & Liusvaara, 2017). Furthermore, I didn't assess my implementation against active adversary attacks, which is a major concern in real-world scenarios.

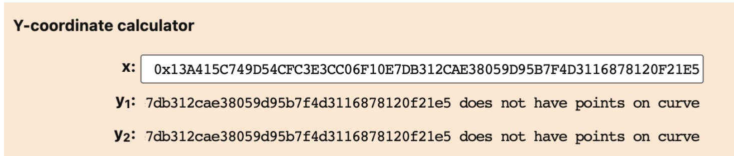
5.2 Observations and Uncertainties

During testing, I observed that the second single-shot test vector from RFC 7748 did not succeed when using the double-and-add algorithm, resulting in an exception indicating that the given x-coordinate is not a valid point on Curve25519 (Figure 5.2). Upon validating with a reference[8] implementation, I found that after clearing the MSB of x, the resulting x-coordinate does not correspond to a valid point on Curve25519. In contrast, the Montgomery ladder successfully computes the correct result, highlighting its robustness against malformed public inputs.

Moreover, I was initially unsure whether to use standard integer division instead of field division when halving the scalar during recursion in the double-and-add algorithm. Since the scalar is clamped, it is supposed to be smaller than p, so I ended up using integer division.

[7] as in the double-and-add algorithm

[8] https://x25519.xargs.org/




```
x-coordinate:
    e5210f12786811d3f4b7959d0538ae2c31dbe7106fc03c3efc4cd549c715a493
last byte:
    0x93 -----> 0x13 (MSB cleared)
updated x-coordinate:
    e5210f12786811d3f4b7959d0538ae2c31dbe7106fc03c3efc4cd549c715a413
big-endian format for tool:
    13a415c749d54cfc3e3cc06f10e7db312cae38059d95b7f4d3116878120f21e5
```

Figure 4: Absence of a valid curve point for the RFC 7748 test vector 2.

References

Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017. URL https://www.rfc-editor.org/info/rfc8032.

Martin Kleppmann. Implementing curve25519/x25519: A tutorial on elliptic curve cryptography. 2020. URL https://martin.kleppmann.com/papers/curve25519.pdf.

Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security. RFC 7748, January 2016. URL https://www.rfc-editor.org/info/rfc7748.

TomCrypto. pycurve25519. https://github.com/TomCrypto/pycurve25519, 2013.