Assignment 2: Elliptic curve signatures (Ed25519)

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 an elliptic curve signature scheme called Ed25519 from scratch. My implementation supports both standard point addition and a relatively faster (less costly) version using extended projective coordinates. My implementation closely follows RFC 8032 (Josefsson & Liusvaara, 2017) for the API design and the Hisil et al. (2008) work on twisted Edwards curves for a faster algorithm point addition. The code is publicly available at https://github.com/ayainfida/p79-assignment2-ed25519.

2 Implementation Architecture

The core implementation is split across modular components, briefly introduced here, with key design choices discussed in Section 3.

Field Elements (`field.py`). Defines an integer modulo prime p field, `ModInt`, supporting arithmetic operations.

Point (`point.py`). Implements point addition for the Edwards curve, as well as computing the x-coordinate.

Primitives (`primitives.py`). Defines type safe wrapper classes around raw byte strings for API type safety.

Encoding/Decoding (`encoding.py`). Handles little-endian byte conversion, point compression/decompression, and scalar clamping.

Double-and-Add (`methods.py`). Provides a scalar multiplication algorithm supporting both standard and extended coordinate systems, implemented recursively.

Ed25519 (`ed25519.py`). Exposes a public Ed25519 interface for signing and verification.

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

3 Implementation Design Choices

3.1 Field Representations

Based on the last assignment's feedback and discussion with Martin, this time I designed a two-tier class hierarchy for field arithmetic: an immutable `ModInt` base class, with both `FieldElement` (modulo $p = 2^{255} - 19$) and `Field_q` (modulo $q \approx 2^{252}$) inheriting from it. The `ModInt` class ensures that every value is reduced modulo its prime during initialization.

As both classes represent $\mathbb{F}_p$ and $\mathbb{F}_q$, I fixed the corresponding prime during initialization, allowing for a single-argument construction. Using separate classes distinguishes both fields at the type level and avoids using the incorrect modulus during scalar multiplication, signing, and verification.

The `ModInt` class supports operator overloading, making the code much more interpretable and less prone to arithmetic errors compared to my last assignment, where I had separate functions for field arithmetic (e.g., `fadd`). This is evident from the following example:

$$\frac{x_1 y_2 + x_2 y_1}{1 + d x_1 x_2 y_1 y_2}$$

```
(A1): fdiv(fadd(fmul(x_1, y_2), fmul(x_2, y_1)),
           fsub(1, fmul(d, fmul(x_1, fmul(x_2, fmul(y_1, y_2))
(A2): ((x_1 * y_2) * (x_2 * y_1)) / (1 + d * x_1 * x_2 * y_1 * y_2)
```

I also implemented explicit field compatibility checks before every arithmetic operation, raising an error if two elements belong to different fields.

```
def __is_field_element(self, other: object):
    if not isinstance(other, ModInt):
        raise TypeError("Elements must be ModInts.")
    if self.p != other.p:
        raise ValueError("Elements must belong to the same field.")
```

3.2 Point Representations

To represent a point on the Edwards curve $-x^2 + y^2 = 1 + dx^2y^2 \pmod{p}$, I introduced two representations: `Point`[1] and `ExtendedPoint`[2]. The `Point` representation is used primarily at the API boundary (e.g., compression and decompression). Its constructor takes the y-coordinate and the sign bit of x, and reconstructs the corresponding x-coordinate accordingly:

$$x = \pm\sqrt{\frac{y^2 - 1}{dy^2 + 1}}$$

To compute the square root, I implemented a `sqrt` operation for `ModInt` using the exponentiation-based method applicable to primes $p \equiv 5 \pmod{8}$, as described in RFC 8032 (Josefsson & Liusvaara, 2017). Additionally, point addition is implemented as:

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1 y_2 + x_2 y_1}{1 + d x_1 x_2 y_1 y_2}, \frac{x_1 x_2 + y_1 y_2}{1 - d x_1 x_2 y_1 y_2}\right)$$

As this formula is complete, no separate doubling formula is required, and is implemented simply as `self + self`. Both x and y are instances of `FieldElement`, so this algebraic expression can be evaluated directly due to operator overloading.

The division operator in `FieldElement` translates into a field inversion followed by multiplication, and since double-and-add repeatedly applies point additions and doublings, this results in many costly inversions. This motivated the introduction of `ExtendedPoint`, where point additions and doublings avoid repeated inversions, deferring a single inversion until conversion back to `Point` form. For `ExtendedPoint`, I followed the addition formulas from Hisil et al. (2008)[3] for the case $a = -1$, which matches the Ed25519 curve equation.

My `ED25519` API supports both standard and fast scalar multiplication through the enum `ED25519ScalarMultAlgorithm`, passed to the `ED25519` constructor. For the latter, the `BASE_POINT`[4] is first converted to `ExtendedPoint`, and the result is converted back to `Point`.

[1] of the form (x, y)
[2] extended projective coordinates $(X : Y : Z : T)$
[3] Twisted Edwards curve is defined as $ax^2 + y^2 = 1 + dx^2y^2$
[4] of the type `Point`

```
if self.algorithm == ED25519ScalarMultAlgorithm.SCALAR_MULT:
    result = double_and_add(k, Pt)
elif self.algorithm == ED25519ScalarMultAlgorithm.FAST_SCALAR_MULT:
    result = double_and_add(k, Pt.to_extended_coordinates())
else:
    raise ValueError(f"Unsupported algorithm: {self.algorithm}")
```

This means that the `double_and_add` function supports both point types:

```
def double_and_add(k: int, Pt: Point | ExtendedPoint) -> Point | ExtendedPoint
```

Since it is implemented recursively, the final result preserves the type of input point.

3.3 Cryptographic Primitives

I really liked the idea of using Type States in Daniel's lecture on *Software Engineering Principles*. These type states enforce correct API usage and preserve logical guarantees throughout the program's lifetime. As a result, for this assignment, rather than passing raw `bytes`, I introduced dataclasses: `Key`, `Message`, and `Signature`. For further distinction between public and private keys, I introduced explicit wrapper types: `PrivateKey` and `PublicKey`. All of these primitive classes are immutable, which prevents accidental mutation during processing.

By introducing distinct types, misuse of raw `bytes`[5] is avoided: a function expecting a `PublicKey` will not accept a `PrivateKey`. This in turn makes function signatures self-documenting: e.g., the signing function requires a `PrivateKey`, while verification requires a `PublicKey`:

```
def sign(self, msg: Message, sk: PrivateKey) -> Signature
def verify(self, msg: Message, sig: Signature, pk: PublicKey) -> bool
```

Furthermore, I also introduced a `LengthError` exception to handle invalid length inputs at the API boundaries, making error messages more user-friendly and self-expressive.

4 Testing and Validation

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

Ed25519 API Testing. Validated my implementation against the test vectors published in RFC 8032 (Josefsson & Liusvaara, 2017) and Project Wycheproof (McCarney et al., 2016). Tested type validation, length validation, as well as unforgeability, authenticity, integrity, and determinism.

Point Addition Agreement. Verified that both standard and fast point addition produce identical results for the same inputs.

Encoding. Tested point compression/decompression round-trip property, MSB masking for y-coordinates and scalar clamping (Josefsson & Liusvaara, 2017).

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] At the very core, key, signature, and message are of type `bytes`

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. Moreover, I have implemented the double-and-add algorithm in a recursive fashion that has data-dependent branches, which could potentially leak scalar bits. For a production-quality implementation, constant-time field arithmetic, uniform memory accesses, and avoidance of data-dependent branches[6].

In addition to these timing attacks, there are a few validation gaps in my implementation. I did not explicitly check that a decoded public key is in the prime-order subgroup. Similarly, point decompression stores the encoded y-coordinate as a `FieldElement`, reducing it modulo p, thereby not rejecting non-canonical encodings. These were fine to not be considered for assignment purposes, but production-quality requires stricter validation.

5.2 Observations and Uncertainties

During testing, I intentionally passed invalid types to the `verify` and `sign` methods to ensure that a `TypeError` is raised. However, this failed the static type checker [7], since the test itself deliberately violates the function's type signature. To proceed, I added `# type: ignore` to the corresponding lines in the test files. I faced a similar issue with the `double_and_add` function. Given that it supports both `Point` and `ExtendedPoint` types and preserves the return type accordingly at runtime, the static type checker did not recognize this, and I had to explicitly add the ignore comment; otherwise, the code would have been full of instance checks.

According to Hisil et al. (2008), `ExtendedPoint` was supposed to be faster, but I did not see any differences in test case timings; maybe it is noticeable at a finer granularity. I chose to add an `enum` option for selecting the point addition method, which was useful for testing but is not standard practice.

Moreover, I first added length checks in the primitive types, which did not allow invalid byte lengths, but this raised exceptions even before executing the function. Therefore, I removed these checks and explicitly took them into account in my API-exposed functions—*I wonder what the best way to do this is*!

References

Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted edwards curves revisited. In *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 326–343. Springer, 2008.

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

Daniel McCarney, Filippo Valsorda, Samuel Lucas, and Andrew Ayer. Project wycheproof: Test vectors for cryptographic implementations. https://github.com/C2SP/wycheproof, 2016.

[6] similar to the Montgomery ladder, which has a fixed sequence of operations

[7] `uv run ty check`