Lab Report 1 - Diffie-Hellman, Signatures

Yuval Weiss
yw580

February 2025

1 Introduction

1.1 Curve25519 and X25519

Curve25519 is a cryptographic elliptic curve introduced in RFC 7748 [8]. More specifically, it is a Montgomery curve [2] of the form

$$v^2 = u^3 + Au^2 + u \tag{1}$$

with an underlying field defined by a prime p. For Curve25519, $p = 2^{255} - 19$ and $A = 486662$. The curve provides a 128-bit security level. RFC 7748 defines the `X25519` as the scalar multiplication of a curve point. The multiplication is performed on the u-coordinate only and outputs the resulting scalar product's u-coordinate. `X25519` can also be used as the base of a Diffie-Hellman key-exchange protocol [5], wherein a secret scalar is chosen to multiply a fixed base point, thus generating a public key. Each person's public key is now scalar multiplied by the other's secret scalar, resulting in a shared secret. This procedure is displayed in Algorithm 1 below.

Algorithm 1 X25519-based Diffie-Hellman protocol

Require: `X25519` scalar multiplication function
Require: $a \leftarrow$ 32 randomly chosen bytes

function DIFFIE-HELLMAN(B) ▷ key-exchange partner B
  $K_A \leftarrow \texttt{X25519}(a, 9)$ ▷ assume `X25519` normalises a
  send $K_A \rightarrow B$
  receive $K_B \leftarrow B$ ▷ K_B generated analogously to K_A
  $K \leftarrow \texttt{X25519}(a, K_B)$ ▷ shared secret, B computes $\texttt{X25519}(b, K_A) = K$
  return K

Ensure: must not use K as an encryption key directly, instead use a key-derivation function based on K, K_A and K_B

1.2 Ed25519

RFC 8032 defines the Edwards-Curve Digital Signature Algorithm (EdDSA) on the edwards25519 curve [6], which is equivalent to Curve25519 under a change of coordinates. This algorithm uses both coordinates, conventionally denoted x and y. The two components of the algorithm, the `sign` and `verify` operations, are described in Algorithm 2 below.

Algorithm 2 The `sign` and `verify` operations for Ed25519

Require: $\textsc{Compress}(p)$ ▷ converts point $p = (x, y)$ to a 32-byte representation
Require: $\textsc{Decompress}(b)$ ▷ inverse of Compress, can return $\varnothing$ (null)
Require: $\textsc{Expand}(s')$ ▷ splits s' into s and *prefix*, and clamps s
Require: $\textsc{Sha512}(c)$ ▷ returns `Sha512` hash of byte contents c as an integer
Require: $\mathcal{B}$ ▷ base point of edwards25519 curve
Require: p, q ▷ curve prime p, group order q

function $\textsc{Sign}(secret, message)$
  $s, prefix \leftarrow \textsc{Expand}(secret)$
  $p_k \leftarrow \textsc{Compress}(s \cdot \mathcal{B})$ ▷ $\cdot$ denotes scalar multiplication of curve point
  $r \leftarrow \textsc{Sha512}(prefix \parallel message) \mod q$ ▷ $\parallel$ denotes byte concatenation
  $R \leftarrow \textsc{Compress}(r \cdot \mathcal{B})$
  $k \leftarrow \textsc{Sha512}(R \parallel p_k \parallel message) \mod q$
  $t \leftarrow r + k \times s \mod q$ ▷ $\times$ denotes integer multiplication
  return $R \parallel t$ ▷ signature

function $\textsc{Verify}(pub, msg, sig)$ ▷ public key pub, message msg, signature sig
  if $\#pub \neq 32$ **then** ▷ $\#$ denotes number of bytes
    throw invalid public key length
  if $\#sig \neq 64$ **then**
    throw invalid signature length
  $A \leftarrow \textsc{Decompress}(pub)$
  if $A = \varnothing$ **then**
    return false
  $R \leftarrow \textsc{Decompress}(sig_{[:32]})$ ▷ first 32 bytes of sig
  if $R = \varnothing$ **then**
    return false
  $t \leftarrow sig_{[32:]}$ ▷ last 32 bytes of sig
  if $t \geq q$ **then**
    return false
  $k \leftarrow \textsc{Sha512}(sig_{[:32]} \parallel pub \parallel msg) \mod q$
  return $t \cdot B \stackrel{?}{=} R + k \cdot A$ ▷ curve point addition & scalar multiplication

2 Implementation

2.1 Outline and approach

The code was structured with five main classes, as shown in Listing 1. X25519Base is an abstract base class in which core functions for Curve25519 are implemented, namely encoding/decoding functionality; the Montgomery ladder [9]; and the standard double-and-add. Ed25519Base is the corresponding abstract base class, containing core functionality for implementing the Ed25519 algorithm. As a result of being abstract, both base classes contain only static (stateless) methods. The Curve25519 *frozen* (immutable) dataclass stores constants characterising the equivalent curves used in X25519 and Ed25519. Furthermore, the base classes define constants such as BYTE_ORDER and ALLOWED_LEN to prevent typos and the use of so-called 'magic numbers'.

```
@dataclass(frozen=True)
class Curve25519:
    '''Holds constants related to Curve25519'''

class X25519Base(abc.ABC):
    '''Abstract base class implementing core Curve25519 functions'''

class X25519Client(X25519Base):
    '''Concrete client-facing implementation of Diffie-Hellman using Curve25519'''

class Ed25519Base(abc.ABC):
    '''Abstract class implementing core Ed25519 functions'''

class Ed25519Client(Ed25519Base):
    '''Concrete client-facing implementation of Ed25519'''
```

Listing 1: Outline of classes used to implement X25519 and Ed25519 functionality

Two client-facing classes are defined: X25519Client and Ed25519Client. Both expose simple APIs and defer practically all calculation details to the abstract superclasses they inherit from.

More generally, explicit type annotations were used (and checked with mypy) throughout the codebase to minimise the opportunity for any human coding errors.

2.2 Curve25519

As outlined in Section 2.1 above, Curve25519 is a frozen dataclass holding curve constants p, d, q and A. It implements a single method, mod_mult_inv, which computes the multiplicative inverse modulo p and is shown in Listing 2.

2.3 X25519

X25519Base implements two methodologies for calculating X25519: the Montgomery ladder [2]; and a reformulation of the ladder based on an explicit double-and-add

```python
@dataclass(frozen=True)
class Curve25519:
    # curve constants p, d, q, A, a24 defined here...
    @staticmethod
    def mod_mult_inv(x: int) -> int:
        return pow(x, Curve25519.p - 2, Curve25519.p)
```

Listing 2: Modular multiplicative inverse for Curve25519

method using projective (u, z)-coordinates. Furthermore, it defines primitives for encoding/decoding scalars and u-coordinates to and from integers and bytes/hex-strings. The ladder method is based predominantly on the model implementation in RFC 7748 [8], while, the double-and-add variant is based on the rest of the implementation in [10].

A type, `DecodeInput`, was introduced to maximise the flexibility of the scalar and u-coordinate decoding methods. This had an associated `_decode_input_to_list_int` function (shown alongside the type in Listing 3) to canonicalise inputs to the decoding functions.

```python
type DecodeInput = str | list[int] | bytes | int

@staticmethod
def _decode_input_to_list_int(x: DecodeInput) -> list[int]:
    ALLOWED_LEN = X25519Base.ALLOWED_LEN
    def validate_length(c: bytes | list[int]) -> None:
        if (length := len(c)) != ALLOWED_LEN:
            raise DecodeSizeError(ALLOWED_LEN, length)
    if isinstance(x, str):
        bs = bytes.fromhex(x)
        validate_length(bs)
        return list(bs)
    if isinstance(x, list):
        validate_length(x)
        return [z & 0xFF for z in x]
    if isinstance(x, int):
        try:
            return list(x.to_bytes(ALLOWED_LEN, X25519Base.BYTE_ORDER))
        except OverflowError as e:
            raise DecodeSizeError(ALLOWED_LEN, (x.bit_length() + 7) // 8) from e
    validate_length(x) # x must be bytes
    return list(x)
```

Listing 3: Canonicalisation of input types for decoding to scalars or u-coordinates

Both versions utilise a constant time swap method based on the one in [10], displayed in Listing 4. This constant-time swap method is more Pythonic than the one proposed in the RFC and has the advantage of being type-generic.

The final implementation of `X25519Client`, displayed in Listing 5, leverages the

```python
@staticmethod
def _const_time_swap[T](a: T, b: T, swap: int) -> tuple[T, T]:
    index = int(swap) * 2
    temp = (a, b, b, a)
    return temp[index], temp[index + 1]
```

Listing 4: Constant time swap method modelled on the version in [10]

RFC-based Montgomery ladder implementation [8], which demonstrated superior performance in informal testing. The inheritance abstractions ensure that the class is minimal, enabling users with no exposure to the field to use it as a cryptographic tool[1].

```python
class X25519Client(X25519Base):
    type Key = str
    BASE_POINT_U = X25519Base._decode_u_coordinate('09' + 31 * '00')

    _private: int
    _public: int
    _public_hex_str: Key

    def __init__(self, secret: DecodeInput | None = None) -> None:
        if secret is None:
            secret = random(self.ALLOWED_LEN)
        self._private = self._decode_scalar(secret)

        # derive public key
        self._public = self._compute_x25519_ladder(self._private, self.BASE_POINT_U)
        self._public_hex_str = self._encode_u_coordinate(self._public, to_str=True)

    @property
    def public(self) -> Key: # public key get method
        return self._public_hex_str

    def compute_shared_secret(self, other_pk: Key, *, abort_if_zero: bool = False):
        shared_secret = self._compute_x25519_ladder(self._private, other_pk)
        if abort_if_zero and shared_secret == 0:
            raise ZeroSharedSecret('Shared secret was 0, aborting!')
        return shared_secret
```

Listing 5: Client-facing implementation of X25519-based Diffie-Hellman key exchange

[1]With the caveat that the code was modified and verified extensively to ensure production-level cryptographic standards.

```python
class Ed25519Point:
    X: int; Y: int; Z: int; T: int

    @classmethod
    def neutral_element(cls) -> 'Ed25519Point':
        # returns neutral element (0, 1, 1, 0)

    @classmethod
    def base_point(cls) -> 'Ed25519Point':
        # returns base point G

    def __add__(self, Q: 'Ed25519Point') -> 'Ed25519Point':
        # add points P and Q, called with P + Q

    def double(self) -> 'Ed25519Point':
        # Double point p, which saves some operations over adding to itself

    def __mul__(self, s: int) -> 'Ed25519Point':
        # multiply P by scalar s, called with P * s

    def __rmul__(self, s: int) -> 'Ed25519Point':
        # allows scalar multiplication to be called with s * P

    def __eq__(self, Q: object) -> bool:
        # uses the identity x1 / z1 == x2 / z2 <=> x1 * z2 == x2 * z1
        # called with P == Q
```

Listing 6: `Ed25519Point` – main arithmetic primitives

2.4 Ed25519

A class, `Ed25519Point`, implements curve point arithmetic to be used in the Ed25519 calculation primitives defined in `Ed25519Base`. The class represents curve points in extended homogeneous coordinates, wherein a point (x, y) is defined as in Equation 2.

$$\begin{gathered}(x, y) \text{ is represented as } (X, Y, Z, T) \\ \text{where} \\ x = \frac{X}{Z}, \quad y = \frac{Y}{Z}, \quad x \cdot y = \frac{T}{Z}\end{gathered} \tag{2}$$

`Ed25519Point`'s implementation of curve pointarithmetic primitives, based on the Python illustration in RFC 8032 [6], is displayed in Listing 6. The class implements the `__add__`, `__mul__` and `__eq__` magic methods, enabling clearer calculation functions in `Ed25519Base`. Two constant points are defined: the neutral element $(0, 1, 1, 0)$ and the base point according to the formula in the RFC. The class implements further primitives for point compression and decompression, outlined in Listing 7.

Defining magic methods on `Ed25519Point` enables the implementations of the `sign` and `verify` operations in `Ed25519Base` to be much clearer. The operations are aligned to their definitions in the lecture slides [7]. `Ed25519Base` also has a primitive for expanding a secret from `bytes` format to s_{bits} and *prefix*, as Listing 8 shows.

```python
class Ed25519Point: # continued...
    @staticmethod
    def recover_x(y: int, sign: int) -> int | None:
        if y >= Curve25519.p:
            return None
        x2 = (y * y - 1) * Curve25519.mod_mult_inv(Curve25519.d * y * y + 1)
        if x2 == 0:
            return None if sign else 0
        x = pow(x2, (Curve25519.p + 3) // 8, Curve25519.p) # find square root of x2
        if (x * x - x2) % Curve25519.p != 0:
            x = x * Ed25519Base.modp_sqrt_m1 % Curve25519.p
        if (x * x - x2) % Curve25519.p != 0:
            return None
        return x if x & 1 == sign else Curve25519.p - x

    def compress(self) -> bytes:
        z_inv = Curve25519.mod_mult_inv(self.Z)
        x = self.X * z_inv % Curve25519.p  # equivalent to X / Z
        y = self.Y * z_inv % Curve25519.p  # equivalent to Y / Z
        res = y | ((x & 1) << 255)
        return res.to_bytes(Ed25519Base.KEY_LEN, Ed25519Base.BYTE_ORDER)

    @classmethod
    def decompress(cls, s: bytes) -> 'Ed25519Point | None':
        if len(s) != Ed25519Base.KEY_LEN:
            raise DecompressionError(Ed25519Base.KEY_LEN, len(s))
        y = int.from_bytes(s, Ed25519Base.BYTE_ORDER)
        sign = y >> 255
        y &= (1 << 255) - 1
        x = cls.recover_x(y, sign)
        return cls(x, y, 1, x * y % Curve25519.p) if x else None
```

Listing 7: `Ed25519Point` – point compression and decompression primitives

```python
class Ed25519Base(abc.ABC):
    KEY_LEN: Final = 32
    SIG_LEN: Final = 64
    BYTE_ORDER: Literal['little'] = 'little'

    @staticmethod
    def _sha512(s: bytes): return hashlib.sha512(s).digest()

    @staticmethod
    def _secret_expand(secret: bytes) -> tuple[int, bytes]:
        if len(secret) != Ed25519Base.KEY_LEN:
            raise BadKeyLengthError(Ed25519Base.KEY_LEN, len(secret))
        h = Ed25519Base._sha512(secret)
        a = int.from_bytes(h[: Ed25519Base.KEY_LEN], Ed25519Base.BYTE_ORDER)
        a &= (1 << 254) - 8
        a |= 1 << 254
        return (a, h[Ed25519Base.KEY_LEN :])
```

Listing 8: `Ed25519Base` – secret expansion and definition of constants

```python
class Ed25519Client(Ed25519Base): # client facing implementation
    _secret: bytes
    _public: bytes
    type ClientInput = bytes | str

    def __init__(self, secret: ClientInput | None) -> None:
        if secret:
            self._secret = self._clean_input(secret)
            if len(self._secret) != self.KEY_LEN:
                raise BadKeyLengthError(self.KEY_LEN, len(self._secret))
        else:
            self._secret = random_bytes(32)
        self._public = self._secret_to_public(self._secret)

    @staticmethod
    def _clean_input(data: ClientInput) -> bytes:
        return bytes.fromhex(data) if isinstance(data, str) else data

    @property
    def public(self):  # getter for public key
        return self._public

    def sign(self, msg: ClientInput) -> bytes:  # sign message
        return self._sign(self._secret, self._clean_input(msg))

    def verify(self, pub: ClientInput, msg: ClientInput, sig: ClientInput) -> bool:
        return self._verify(
            self._clean_input(public),
            self._clean_input(msg),
            self._clean_input(signature),
        )
```

Listing 9: `Ed25519Client` – client-facing implementation for Ed25519

By abstracting the calculation functions to the base class, the final client-facing class, `Ed25519Client` has a highly streamlined realisation, as displayed in Listing 9. The secret and public keys are stored internally as `bytes`, hence the client can be initialised with a hex string or byte array.

2.5 Error handling

All exceptions used throughout the code were defined in the `errors.py` file and are presented in Listing 10. `DecompressionError`, `BadKeyLengthError`, `BadSignatureLengthError` and `DecodeSizeError` inherit from a base class `BadLengthError`, which itself inherits from `ValueError`. `BadLengthError` defines a base message and `__init__` method for the other exception classes to customise. The final error, `ZeroSharedSecret`, is used optionally within the shared secret calculation in `X25519Client`, and is raised when the shared secret is 0. These specialised errors enable clarity for users using the two cryptographic clients when unrecoverable errors are introduced through inputs.

```python
class BadLengthError(ValueError):  # base class for bad lengths
    message_base = ''
    unit = ''
    def __init__(self, exp_len: int, got_len: int) -> None:
        if self.unit != '':
            self.unit = ' ' + self.unit
        msg = f'{self.message_base}, expected {exp_len}{self.unit} but got {got_len}'
        super().__init__(msg)

class DecompressionError(BadLengthError):  # point decompression error
    message_base = 'Error decompressing'

class BadKeyLengthError(BadLengthError):  # key expansion error
    message_base = 'Bad key length'

class BadSignatureLengthError(BadLengthError):  # invalid signature length error
    message_base = 'Bad signature length'

class DecodeSizeError(BadLengthError):  # invalid scalar/u-coordinate size
    message_base = 'Invalid scalar/u-coordinate'

class ZeroSharedSecret(ValueError):  # error for zero shared secret
    def __init__(self) -> None:
        super().__init__('Shared secret was 0, aborting!')
```

Listing 10: `errors.py` – defining exceptions used throughout the program

3 Validation

Validity was ensured across two domains: code correctness and accurate computation according to the RFC specifications. Static type annotations were incorporated to improve code clarity and maintainability, functioning alongside `mypy` which detected potential type-related errors throughout development. This approach minimised the risk of invalid operations and facilitated early detection of inconsistencies, ultimately increasing the robustness of the implementation.

The accuracy of the implementation was verified through unit testing with `pytest`. Tests were holistic: targeting everything from primitives in the base classes to abstracted methods in the client classes, with 32 unique test methods comprising 2962 unique test cases. The overall code coverage of the unit tests was 98%.

`X25519` and `Ed25519` were both tested using their respective RFC test vectors [2, 8]. For the former protocol, this included looping the ladder calculation for 1, 1000 and 1,000,000[2] iterations and comparing the result to the published test cases. The RFC test vectors were further bolstered by larger test vector files. For `X25519`, the test vectors published by Project Wycheproof [4] were used,[3] while for `Ed25519` the supplementary test vectors were sourced from Bernstein's website [3].[4]

[2] The 1,000,000 iteration case is commented out in the submission as it has a runtime of over 20 minutes.

[3] Wycheproof test vectors for `X25519` were sourced here: `https://github.com/C2SP/wycheproof/blob/master/testvectors/x25519_test.json`.

[4] Bernstein's test vectors for `Ed25519` were found here: `http://ed25519.cr.yp.to/python/sign`.

Further tests included checks to ensure that the appropriate exceptions are raised at the appropriate time, and also to check the code's handling of edge cases.

4 Findings

Implementing these cryptographic protocols presented challenges and learning opportunities. An early version of the code did not compute mods for intermediate calculation steps, which led to severely impaired performance. Introducing the relevant mod operations at each step significantly improved efficiency, highlighting the trade-off between managing the size of integers and the additional mod operation overhead.

Another mode of learning was found in the consideration of how to store keys internally. For `Ed25519`, the code lent itself more obviously to storing keys in `bytes` format, but for `X25519`, the case was more nuanced due to the structure of the model code in RFC 7748 [2]. While storing hex strings fit the code most easily, this seemed inefficient. Therefore, `ints` were chosen as the internal storage type, which would remove the need for encoding/decoding to and from strings and integers.

5 Productionisation

While the implementations of X25519 and Ed25519 achieve functioning correct key exchange and digital signature algorithms, additional steps must be taken to ensure industry-level security, performance and compliance.

Production-quality implementations must be resistant to side-channel attacks such as timing analysis. For this reason, all operations must be constant-time. Since this implementation leverages Python's built-in big int handling, which does *not* run in constant time, these implementations cannot be classified as cryptographically secure.

Likewise, memory management must be carefully handled, and sensitive data must be securely handled and deleted after use. It might also be beneficial to programmatically prevent the re-use of a secret key for multiple sessions. Currently, secrets are stored directly in the class, but meticulous key security must be implemented for production-quality code.

Error handling methodologies are also sensitive to exploitation. Unintentional information leaks must be prevented, and error handling and messaging must adhere strictly to industrial guidance.

Efficient implementations are just as vital as security concerns, and pure Python programs probably do not achieve this benchmark. Low-level optimisations are ubiquitous in production-level code, even at the cost of clarity of implementation.

Furthermore, code auditing and third-party verification are vital to creating truly secure cryptographic implementations. Common vulnerabilities, such as improper key derivation or inadequate entropy sources, can be identified and eliminated through stringent review processes. Formal verification techniques such as theorem proving and symbolic execution provide even higher levels of assurance of the correctness of

---

`input`

cryptographic implementations. Tools such as Tamarin [1] can be used to achieve rigorous verification of cryptographic protocol validity.

References

[1] David Basin et al. "Tamarin: Verification of Large-Scale, Real World, Cryptographic Protocols". In: *IEEE Security and Privacy Magazine* (2022). DOI: 10.1109/msec.2022.3154689. URL: https://hal.science/hal-03586826.

[2] Daniel J. Bernstein and Tanja Lange. *Montgomery Curves and the Montgomery Ladder*. 2017. URL: https://eprint.iacr.org/2017/293. Pre-published.

[3] Daniel J. Bernstein et al. *Ed25519: High-Speed High-Security Signatures*. 2017. URL: http://ed25519.cr.yp.to/index.html.

[4] Daniel Bleichenbacher et al. *C2SP/Wycheproof*. Community Cryptography Specification Project, Feb. 13, 2025. URL: https://github.com/C2SP/wycheproof.

[5] W. Diffie and M. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654. ISSN: 1557-9654. DOI: 10.1109/TIT.1976.1055638. URL: https://ieeexplore.ieee.org/document/1055638/?arnumber=1055638.

[6] Simon Josefsson and Ilari Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. Request for Comments RFC 8032. Internet Engineering Task Force, Jan. 2017. 60 pp. DOI: 10.17487/RFC8032. URL: https://datatracker.ietf.org/doc/rfc8032.

[7] Martin Kleppmann and Daniel Hugenroth. "P79: Cryptography and Protocol Engineering". Lecture Slides (University of Cambridge). 2025. URL: https://www.cl.cam.ac.uk/teaching/2425/P79/p79-slides.pdf.

[8] Adam Langley, Mike Hamburg, and Sean Turner. *Elliptic Curves for Security*. Request for Comments RFC 7748. Internet Engineering Task Force, Jan. 2016. 22 pp. DOI: 10.17487/RFC7748. URL: https://datatracker.ietf.org/doc/rfc7748.

[9] Peter L. Montgomery. "Speeding the Pollard and Elliptic Curve Methods of Factorization". In: *Mathematics of Computation* 48.177 (1987), pp. 243–264. ISSN: 00255718, 10886842. JSTOR: 2007888. URL: http://www.jstor.org/stable/2007888.

[10] Nicko van Someren. *A Pure Python Implementation of Curve25519*. Gist. 2021. URL: https://gist.github.com/nickovs/cc3c22d15f239a2640c185035c06f8a3.