Cryptography and Protocol Engineering (P79)
Assignment 1

Firas Aleem *(faa42)*

Lent 2024

Word Count: 1,989

1 Assignment 1: Curve25519 Diffie-Hellman and Ed25519 Signatures

X25519 and Ed25519 are widely used cryptographic primitives based on elliptic curve cryptography (ECC). X25519 is primarily used for key exchange in secure communication protocols such as TLS and Signal, while Ed25519 is a digital signature scheme designed for cryptographic signing. Both rely on finite field arithmetic and use the Curve25519 structure to ensure strong cryptographic properties.

This report explores my implementation of X25519 and Ed25519, focusing on design decisions, correctness, and performance.

- **Section 2:** Covers X25519, including key exchange, scalar multiplication strategies, and testing.
- **Section 3:** Discusses Ed25519, detailing signature generation, encoding constraints, and batch verification.
- **Section 4:** Examines findings, performance trade-offs, and optimizations for production readiness.

The report also compares performance against *PyNaCl* [1], highlighting areas for improvement.

2 X25519: Implementation and Testing

2.1 Overview and API Design

To implement X25519, I designed a wrapper class, `X25519`, which allows the user to choose between two different methods for scalar multiplication: the Montgomery ladder and the double-and-add algorithm. This approach provides several advantages:

- **Modularity:** If additional scalar multiplication methods are required, they can be added as new options in the wrapper without affecting the existing implementations.
- **Encapsulation:** The details of whether Montgomery ladder or double-and-add is used are abstracted away from the user, who calls `scalar_multiply`, with either `ladder` or `double-and-add` as an argument.
- **Code Maintainability:** Since both implementations are separate, changes to one do not affect the other, ensuring cleaner and more maintainable code.
- **Shared Functionality:** Common operations, such as private key clamping and key generation, are handled at the wrapper level, reducing redundancy.

The `X25519` class exposes a method `scalar_multiply`, which internally selects the appropriate method based on how the instance was initialized. It also provides methods for generating private and public keys, ensuring that both implementations are consistent.

2.2 Double-and-Add Implementation

The double-and-add algorithm is implemented in the `MontgomeryDoubleAdd` class. This method follows a standard left-to-right binary approach, where the scalar is iteratively processed, and the point is either doubled or added depending on the least significant bit of the scalar.

Type Considerations: To keep the implementation straightforward and modular, I used:

- `Point` type: Defined as a tuple of two integers `(x, y)`, with `None` used to represent the point at infinity.
    - Using tuples provides a clear and simple representation of elliptic curve points while ensuring immutability.
    - The use of `None` to denote the point at infinity avoids the need for a separate class or special case handling, making operations such as addition and doubling more intuitive.
    - This approach follows the common mathematical notation where the point at infinity is treated as a special case.
- Private and public keys as `bytes`:
    - Keeping keys as `bytes` aligns with how they are represented in cryptographic standards (e.g., RFC 7748).
    - This avoids ambiguity when handling different representations (big-endian vs. little-endian integers) and ensures direct compatibility with standard cryptographic libraries.
    - The conversion functions for transforming between `bytes` and integers are centralized in utility functions, ensuring correctness and reducing redundant logic across different implementations.

Implementation Details: The `add` and `double` methods follow the standard Montgomery curve formulas:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \mod p \tag{1}$$

$$x_3 = \lambda^2 - A - x_1 - x_2 \mod p \tag{2}$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \mod p \tag{3}$$

For point doubling:

$$\lambda = \frac{3x_1^2 + 2Ax_1 + 1}{2y_1} \mod p \tag{4}$$

$$x_3 = \lambda^2 - A - 2x_1 \mod p \tag{5}$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \mod p \tag{6}$$

The `scalar_multiply` function iterates through the bits of the scalar, applying doubling at each step and adding when the corresponding bit is set. I implemented these according to the standard formulas which were found in the lecture notes [2], as well as the Explicit-Formulas Database [3].

2.3 Montgomery Ladder Implementation

The Montgomery ladder is implemented in the `MontgomeryLadder` class. Unlike the double-and-add method, the Montgomery ladder operates entirely in *projective coordinates*. This was implemented using RFC 7748 [4] and Martin's Curve25519 tutorial [5].

Affine and Projective Coordinates: The ladder method maintains two projective points, `(x2, z2)` and `(x3, z3)`, and iterates over the scalar bits using a structured ladder step function:

$$A = (x_2 + z_2)^2 \mod p \tag{7}$$
$$B = (x_2 - z_2)^2 \mod p \tag{8}$$
$$E = A - B \mod p \tag{9}$$
$$C = (x_3 + z_3)(x_2 - z_2) \mod p \tag{10}$$
$$D = (x_3 - z_3)(x_2 + z_2) \mod p \tag{11}$$

At each iteration, the points are swapped conditionally using a *constant-time swap function* to avoid branching-based timing leaks.

Implementation Design Choices:

- **Type Usage:** The `Point` type is still a tuple, but in this case, `y` is always `None`, as X25519 only operates on the x-coordinate.
- **Step Function for Clarity:** The ladder step function is extracted into a separate method for readability and modularity.
- **Constant-Time Execution:** The `constant_swap` function is used to prevent side-channel attacks.

2.4 Utility Functions and Modularity

To maintain a clean and modular structure, I implemented a separate utilities file, which contains:

- **Mathematical functions**: Multiplicative inverse, field addition, multiplication, and square root modulo p.
- **Byte-integer conversions**: Conversions between byte representations and integer values.
- **y-coordinate calculation**: Required for the double-and-add implementation.

By centralizing these operations, I ensured that both scalar multiplication methods (and later the Ed25519 class) could use them without redundancy. Additionally, this approach simplifies testing and debugging, as the utility functions can be independently tested and if needed, could be replaced with optimized versions.

2.5 Testing and Debugging

To ensure correctness and robustness, I implemented a comprehensive test suite covering:

- Unit tests for individual scalar multiplication implementations (Montgomery Ladder and Double-and-Add).
- Validation against RFC 7748 test vectors to ensure compliance with standards.
- Comparisons against PyNaCl to verify correctness.
- Performance benchmarking to compare execution times.
- ECDH shared secret validation to ensure interoperability.

2.5.1 Unit Testing

I tested the Montgomery Ladder and Double-and-Add implementations separately to verify core operations.

Montgomery Double-and-Add:

- Addition and doubling were verified to ensure correct behavior, including handling edge cases like identity elements and inverses.
- Scalar multiplication was tested to confirm that the resulting point was still on the curve.

Montgomery Ladder:

- Scalar multiplication was tested against known values and verified for correctness.
- Tested scalar multiplication using large values and compared results with PyNaCl.

2.5.2 Validation Against RFC 7748 and PyNaCl

I validated my implementation using RFC 7748 test vectors:

- **Vector 1** passed for both implementations.
- **Vector 2** failed for the Double-and-Add method, as expected, because it requires computing a square root, and A is not a quadratic residue modulo p. This behavior is consistent with expectations, as the Montgomery Ladder should succeed while Double-and-Add fails.

To further verify correctness, I compared my implementation with PyNaCl:

- Scalar multiplication outputs matched PyNaCl results across 100 iterations.
- Public key generation was validated against `crypto_scalarmult_base`.

2.5.3 Elliptic Curve Diffie-Hellman (ECDH) Testing

Using RFC 7748 test vectors, I verified:

- Alice and Bob's public keys were correctly generated.
- Shared secret computations were consistent and non-zero.

2.5.4 Performance Benchmarking

I measured execution times to compare efficiency, averaged over 100 iterations:

Montgomery Ladder vs. Double-and-Add:

- **Montgomery Ladder**: 0.00150 s
- **Double-and-Add**: 0.05100 s
- **Speedup**: Ladder is around **36x** faster.

My Implementation vs. PyNaCl:

- **My X25519 (Ladder)**: 0.00146 s
- **PyNaCl**: 0.000124 s
- **Speedup**: PyNaCl is around **12x** faster than my implementation, as expected for a highly optimized cryptographic library.

2.6 Summary

Through unit tests, RFC 7748 validation, PyNaCl comparisons, and performance benchmarking, I confirmed:

- Correctness of both Montgomery Ladder and Double-and-Add.
- Expected behavior in edge cases.
- ECDH shared secret computations were valid.
- Significant efficiency gains with the Montgomery Ladder method, as expected. Also as expected, the Ladder method was slower than PyNaCl due to PyNaCl's optimized implementation.

3 Ed25519: Digital Signature Scheme

Ed25519 is a signature scheme based on the Edwards-curve Digital Signature Algorithm (EdDSA). This implementation follows RFC 8032 [6] and the work of Hisil et al. [7]. Since it operates over a twisted Edwards curve, the most efficient approach is to use extended coordinates, first introduced in that paper [7].

3.1 Coordinate Representation and Utilities

To efficiently perform operations, I used:

- **Extended coordinates** (X, Y, Z, T) with $Z = 1$ and $T = x \cdot y \mod p$.
- **Affine-to-extended** and **extended-to-affine** conversions.
- **Point encoding/decoding** following RFC 8032, where the y-coordinate is stored and the most significant bit encodes the sign of x.

3.1.1 Point Addition, Doubling, and Normalization

Addition and subtraction follow RFC 8032, using explicit formulas that allow efficient computation. Doubling is optimized separately to reuse intermediate values and improve efficiency. To ensure consistency, all extended points are normalized to $Z = 1$ since multiple representations can exist for the same affine point.

3.2 Key Generation and Signing Process

Key generation and signing follow standard Ed25519 operations:

- **Private key**: 32 random bytes.
- **Public key**: Computed by clamping the private key, multiplying it with the base point, and encoding the result.
- **Signing**:
    1. Compute $H = \text{SHA-512}(\text{private key})$, split into lower 32 bytes and prefix.
    2. Derive the scalar a by clamping the lower 32 bytes.
    3. Compute public key $A = a \cdot B$.
    4. Compute nonce $r = \text{SHA-512}(\text{prefix}||\text{message}) \mod L$.
    5. Compute $R = r \cdot B$ and encode it.
    6. Compute challenge $k = \text{SHA-512}(\text{encode}(R)||\text{encode}(A)||\text{message}) \mod L$.
    7. Compute $S = (r + k \cdot a) \mod L$.
    8. Return signature: $\text{encode}(R)||S$.

3.3 Verification and Batch Verification

Signature verification follows RFC 8032, using the co-factored verification equation:

$$[8]SB = [8]R + [8]kA$$

This choice enables batch verification, implemented following Algorithm 3 from Chalkias, Garillot, and Nikolaenko's paper [8]. Batch verification efficiently checks multiple signatures by accumulating weighted sums of sB, R, and kA, significantly reducing the number of expensive scalar multiplications.

3.3.1 Canonical Encoding Checks

To prevent malleability attacks, the implementation strictly enforces canonical encoding as required by RFC 8032:

- **Canonical S values**: The integer S must be strictly less than L to ensure uniqueness.
- **Canonical R values**: The encoded point R must represent a valid curve point, rejecting malformed encodings.
- **Canonical Public Keys**: The public key must be a valid Ed25519 point and not a non-standard encoding.

Initially, I considered implementing relaxed verification following ZIP215 [9], which allows non-canonical R and A values for backward compatibility. However, this approach contradicts both RFC 8032 and the U.S. Government's NIST FIPS 186-5 standard [10], which require strict adherence to canonical encodings. To maintain compliance with these standards and avoid potential signature malleability issues, I opted for enforcing full canonical encoding checks.

3.4 Testing and Performance Analysis

The implementation was validated through extensive testing:

- **Unit tests** for key generation, encoding/decoding, and signature verification.
- **RFC 8032 test vectors** were used to verify correctness.
- **Edge cases**: Invalid signatures, tampered messages, and large messages (up to 16MB) were tested.
- **Batch verification** was tested for consistency and speed.

3.4.1 Performance Comparison

To evaluate performance, I compared signing and verification times against PyNaCl:

Operation	**Our Time (s)**	**PyNaCl Time (s)**	**Speedup**
Signing (1000 iterations)	0.00405	0.00008	≈50x
Verification (1000 iterations)	0.00451	0.00015	≈30x

Table 1: Performance comparison with PyNaCl

As expected, PyNaCl outperformed my implementation due to its optimized C backend. What was surprising, however, was the relatively smaller speedup of verification compared to signing.

Another comparison was between individual and batch verification. Surprisingly, batch verification was *slower* than individual verification:

- Individual verification (1000 signatures): 4.69 s
- Batch verification (1000 signatures): 6.16 s

This was unexpected, as batch verification should be more efficient in larger batches, but the current implementation may require further optimization. A potential optimization could involve using *Multi-Scalar Multiplication (MSM)*, a technique that optimizes the computation of weighted sums of elliptic curve points. Instead of performing separate scalar multiplications for each signature, MSM computes them simultaneously, reducing redundant operations. Efficient MSM techniques, such as *Straus' algorithm* [11] and *Pippenger's method* [12], precompute partial results and use optimized windowing strategies, which can significantly improve performance.

3.5 Summary

The Ed25519 implementation was successfully tested against RFC 8032, edge cases, and performance benchmarks. While batch verification did not yield the expected speed gains, the implementation remains correct and functional.

4 Findings, Performance, and Analysis

4.1 Challenges and Debugging Insights

One of the most time-consuming debugging challenges involved RFC 7748 test vector 2. The RFC provides the input u-coordinate in both hex and base-10, but my Montgomery Ladder implementation failed when using the hex value. After extensive troubleshooting, I traced the issue to missing masked bits in the scalar computation. A rejected errata suggested this mismatch, but it was not immediately obvious.

This issue underscores how even a single bitmask omission can cause an implementation to fail with certain edge cases, despite passing other tests. It serves as a reminder that cryptographic implementations are highly susceptible to subtle flaws, and correctness cannot be assumed without extensive testing.

4.2 Security Considerations

A key limitation of my implementation is the lack of constant-time execution, making it susceptible to side-channel attacks such as timing analysis. Current arithmetic operations and conditional branching introduce variations in execution time, which could be exploited to extract cryptographic secrets. Additionally, invalid input handling must be hardened to prevent fault-injection attacks that manipulate computations into leaking private information.

For production use, these concerns must be addressed by eliminating data-dependent branching and ensuring constant-time modular arithmetic. Libraries like PyNaCl implement these protections, demonstrating why cryptographic implementations require specialized expertise.

4.3 Production-Readiness: Necessary Improvements

While the implementation is functional, several key improvements are necessary for real-world deployment:

1. **Performance Optimizations:**
   - Implement Multi-Scalar Multiplication (MSM) to speed up batch verification.
   - Integrate precomputed lookup tables to avoid redundant field arithmetic.
   - Use windowing techniques in scalar multiplication for efficiency.
2. **Security Enhancements:**
   - Ensure constant-time execution to prevent timing-based side-channel attacks.
   - Replace conditional branching with bitwise operations for uniform execution.
   - Harden memory access patterns against cache-based attacks.
3. **Interoperability and Compliance:**
   - Validate against additional test vectors beyond RFC 8032.
   - Ensure compliance with NIST FIPS 186-5 for standardization.

5 Conclusion

This project involved implementing and evaluating X25519 for key exchange and Ed25519 for digital signatures, following RFC 7748 and RFC 8032. The implementation was validated using test vectors, unit tests, and comparisons against PyNaCl. While the system is functionally correct, key optimizations, such as Multi-Scalar Multiplication (MSM) for batch verification and constant-time execution for security, are necessary for real-world deployment.

The project highlighted the importance of rigorous testing, as small errors (e.g., missing masked bits) can break cryptographic correctness. Performance benchmarking demonstrated that while this implementation serves an educational purpose, optimized libraries like PyNaCl achieve significantly better performance. Future improvements could focus on further optimizing arithmetic operations and implementing side-channel mitigations.

This work reinforces the broader lesson in cryptographic engineering: correctness is non-trivial, and security requires both theoretical rigor and practical validation.

References

1. Developers, T. P. *PyNaCl: Python binding to the Networking and Cryptography library* Python Package Index (PyPI). Available at: https://pypi.org/project/PyNaCl/ (Accessed: 2025-02-15). 2025 (Page 2).
2. Kleppmann, M. & Hugenroth, D. *P79: Cryptography and Protocol Engineering Lecture Notes* University of Cambridge, MPhil in Advanced Computer Science / Computer Science Tripos, Part III. Lent term 2024/25. Available at: https://www.cl.cam.ac.uk/teaching/2425/P79/ (Accessed: 2025-02-15). 2025 (Page 3).
3. Bernstein, D. J. & Lange, T. *Explicit-Formulas Database: Montgomery curves* https://www.hyperelliptic.org/EFD/g1p/auto-montgom.html. Accessed: 2025-02-15. 2025 (Page 3).
4. Langley, A., Hamburg, M. & Turner, S. *Elliptic Curves for Security* RFC 7748. Jan. 2016 (Page 3).
5. Kleppmann, M. *Implementing Curve25519/X25519: A Tutorial on Elliptic Curve Cryptography* Draft manuscript. Available at: https://martin.kleppmann.com/papers/curve25519.pdf (Accessed: 2025-02-15). Oct. 2020 (Page 3).
6. Josefsson, S. & Liusvaara, I. *Edwards-Curve Digital Signature Algorithm (EdDSA)* RFC 8032. Jan. 2017 (Page 6).
7. Hisil, H. *et al. Twisted Edwards Curves Revisited* Cryptology ePrint Archive, Paper 2008/522. 2008 (Page 6).
8. Chalkias, K., Garillot, F. & Nikolaenko, V. *Taming the Many EdDSAs* in *Security Standardisation Research* (eds van der Merwe, T., Mitchell, C. & Mehrnezhad, M.) (Springer International Publishing, Cham, 2020), 67–90 (Page 7).
9. De Valence, H. *Explicitly Defining and Modifying Ed25519 Validation Rules* Zcash Improvement Proposal (ZIP) 215. Final. Available at: https://zips.z.cash/zip-0215 (Accessed: 2025-02-15). Apr. 2020 (Page 7).
10. National Institute of Standards and Technology (US). *Digital Signature Standard (DSS)* tech. rep. (Washington, D.C., Feb. 2023) (Page 7).
11. Straus, E. G. Addition Chains of Vectors (Problem 5125). *The American Mathematical Monthly* **71,** 806–808 (1964) (Page 8).
12. Pippenger, N. *On the evaluation of powers and related problems* in *Proceedings of the 17th Annual Symposium on Foundations of Computer Science* (IEEE Computer Society, USA, 1976), 258–263 (Page 8).