Cryptography and Protocol Engineering (P79) Assignment 2

Firas Aleem *(faa42)*

Lent 2024

Word Count: 2,047*

*Word count excludes code listings and appendices.

Assignment 2: SIGMA and SPAKE2 Protocols

This report details the implementation of the SIGMA and SPAKE2 protocols using X25519 and Ed25519 for secure key exchange and authentication. SIGMA enables authenticated key exchange with certificates, while SPAKE2 facilitates password-authenticated key exchange (PAKE) without a public key infrastructure.

A key focus of this assignment was designing a structured and consistent cryptographic API. Initially, my X25519 and Ed25519 implementations from Assignment 1 [1] handled raw 32-byte keys, which raised usability and security concerns. As I integrated them into SIGMA and SPAKE2, I refined my API to ensure consistency, usability, and correctness.

The report is structured as follows:

- **Section 1** discusses API refactoring for X25519 and Ed25519, highlighting key improvements.
- **Section 2** explains the SIGMA protocol, including certificate authority (CA) design and secure messaging.
- **Section 3** presents the SPAKE2 implementation and its security considerations.
- **Section 5** explores potential extensions and optimizations.

This project reinforced the importance of practical, well-structured cryptographic code. Iterative improvements led to a more robust and maintainable implementation.

1 Refactoring the Cryptographic API

In my initial implementation for Assignment 1, cryptographic keys were handled as raw 32-byte values. While functional, this approach had drawbacks:

- **Security Risks**: Storing private keys as raw byte arrays increased the risk of accidental exposure.
- **Lack of Structure**: Without a defined key abstraction, operations like key exchange and signing required manual byte management.

To address these issues, I introduced structured key classes for `X25519` and `Ed25519`, encapsulating key-related operations within dedicated objects.

1.1 Ensuring Consistency in API Design

While adding key classes improved structure, inconsistencies emerged when integrating them into the SIGMA and SPAKE2 protocols. For example:

- `Ed25519` had a method `signingKey.generate_verifying_key()`, while
- `X25519` required calling `X25519PublicKey.from_private_key()`.

This inconsistency complicated API usage by requiring different function calls for similar operations. To fix this, I aligned their methods so that public or verifying keys could always be derived from their corresponding private or signing keys.

To refine the design further, I did the following:

- `exchange()` for X25519 to handle Diffie-Hellman key exchange.
- `sign()` and `verify()` for Ed25519, eliminating the need for manual instantiation.

These improvements simplified cryptographic operations and enhanced API usability.[1]

Lessons Learned:

- Designing an API without using it in real-world scenarios led to usability issues that became evident during SIGMA and SPAKE2 integration.
- Maintaining consistency across cryptographic primitives is crucial for usability and maintainability.

This refactoring ultimately resulted in a cleaner, more intuitive cryptographic API, making protocol implementation significantly smoother.

2 Implementing the SIGMA Protocol

2.1 Designing the SIGMA Protocol

My implementation of the SIGMA protocol follows the lecture notes [2] and Section 5.1 of Krawczyk's SIGMA paper [3]. The protocol establishes a secure session between two parties, Alice and Bob, by combining:

- **Key exchange** (X25519) to derive a shared secret.
- **Authentication** (Ed25519) to sign ephemeral keys and verify identities.
- **Message integrity** (HMAC-SHA256) to ensure authenticity.

The handshake consists of four steps across three message exchanges:

1. **Initiator (Alice)** sends her ephemeral public key.
2. **Responder (Bob)** responds with his ephemeral key, certificate, and signature.
3. **Initiator** verifies Bob's response, signs the keys, and sends her certificate.
4. **Responder** verifies Alice's identity and finalizes the session.

To ensure modularity and maintainability, I structured the implementation into four key classes:

1. `SigmaParty` - Manages key pairs and certificates for Alice and Bob.
2. `SigmaHandshake` - Orchestrates the handshake, handling key exchange and authentication.
3. `SigmaKeys` - Derives session and authentication keys using HKDF.
4. `SecureChannel` - Provides encrypted messaging after the handshake via AES-GCM.

This modular approach improves testability, extensibility, and debugging.

[1] After refactoring, all test cases ran successfully (see the `tests_ass1` folder).

2.2 SigmaKeys: Derivation and Security Considerations

The `SigmaKeys` class derives session and authentication keys from the shared secret. Instead of hashing the shared secret with domain separation strings (as in the lecture notes), I used HKDF for stronger key separation and security.

- **HKDF for Key Separation:** Ensures independent keys for session encryption (k_S) and message authentication (k_M), even if the shared secret is reused.
- **Improved Domain Separation:** Longer separation strings further reduce key collision risks.
- **AES-GCM for Encryption:** Similar to TLS cipher suites (e.g., `TLS_AES_128_GCM_SHA256`) but uses 256-bit keys for stronger security.
- **Redundant HMAC-SHA256:** Initially added despite AES-GCM's built-in authentication. Benchmarks showed negligible overhead, so I retained it as an additional integrity check.

2.3 Certificate and CA Design

To simplify implementation, I used a minimal JSON-based certificate format instead of X.509:

```
{
    "subject_name": "Alice",
    "subject_key_b64": "<base64-encoded key>",
    "issuer_name": "TestCA",
    "signature_b64": "<base64-encoded signature>"
}
```

This approach avoids unnecessary complexity in the implementation and implements the key features needed in a certificate, such as the subject's name, public key, issuer, and signature. It could be extended in the future to include additional fields like validity dates and certificate revocation lists.

Each `SigmaParty` obtains a certificate from a `CertificateAuthority` , which issues and verifies them.

```
    # Step 1: Setup CA and create parties
    ca = CertificateAuthority("TestCA")
    alice = SigmaParty("Alice", ca.public_key)
    bob = SigmaParty("Bob", ca.public_key)

    # CA issues certificates
    alice_cert = ca.issue_certificate("Alice", alice.ed25519_public)
    bob_cert = ca.issue_certificate("Bob", bob.ed25519_public)

    # Each party sets its certificate
    alice.set_certificate(alice_cert)
    bob.set_certificate(bob_cert)
```

```

    # Verification
    assert ca.verify_certificate(alice_cert) == True
```

2.4 SIGMA Handshake Implementation

Once the parties are set up, the handshake follows these four steps, closely following the paper and lecture notes:

```
    # Step 2: Perform SIGMA handshake
    handshake = SigmaHandshake(alice, bob)

    # Initiator sends SIGMA_INIT
    sigma_init_msg = handshake.create_initiation_message()

    # Responder processes and responds with SIGMA_RESP
    sigma_resp_msg = handshake.handle_initiation_message(sigma_init_msg)

    # Initiator verifies and sends SIGMA_FINAL
    sigma_final_msg = handshake.process_response_message(sigma_resp_msg)

    # Responder finalizes handshake
    session_key = handshake.finalize_handshake(sigma_final_msg)

    assert session_key == alice.session_key == bob.session_key
```

2.5 Extending to SIGMA-I: Identity Protection

I extended my implementation to support SIGMA-I (Section 5.2 of [3]), which encrypts identities using a derived encryption key (k_E) to protect against passive attackers.

- A new `identity_protection` flag determines whether identities are encrypted.
- The k_E key is derived via HKDF and used for AES-GCM encryption of certificates and signatures.
- Due to the modular design, this required minimal changes, and as benchmarks show, it adds almost no overhead.

2.6 Constant-Time Comparisons

In my initial implementation, I used a `zip + XOR` approach for constant-time HMAC comparison. However, CPython's internal optimizations may introduce unintended timing variations, which can weaken the constant-time property. As a result, I used `hmac.compare_digest()` to ensure secure HMAC verification, as recommended in [4].

While my implementation isn't fully constant-time, applying these methods where feasible strengthens security with minimal effort.

2.7 Testing and Validation

I wrote extensive tests to verify correctness, including:

- **Certificate verification:** Ensures only valid certificates pass.
- **Key exchange correctness:** X25519 shared secret must match.
- **SIGMA handshake validation:** Session keys must be consistent.
- **Secure messaging:** Ensures message integrity and authentication.
- **Attack resistance:** Tests tampering with signatures, MACs, and ciphertexts.

All tests passed successfully, demonstrating the correctness and robustness of the SIGMA implementation.

2.8 Performance Benchmarking

To evaluate efficiency, I benchmarked key operations using `timeit` with warm-up runs to reduce cache effects.

SIGMA vs. SIGMA-I Handshake On average, the SIGMA handshake takes 37ms, while SIGMA-I is slightly *faster*, despite the added encryption. This was unexpected, but the negligible AES-GCM overhead likely explains the minimal difference, especially when compared to the more expensive cryptographic operations. This trend remained consistent, even across multiple benchmark runs.

Metric	SIGMA (ms)	SIGMA-I (ms)
Average Time	37.45	37.30
Median Time	37.45	37.31
Min Time	37.11	37.25
Max Time	37.92	37.34
P90 Latency	38.05	37.36
P99 Latency	38.22	37.37

Table 1: SIGMA vs. SIGMA-I Handshake Benchmark

The minimal difference in execution time suggests that the AES-GCM encryption for identity protection in SIGMA-I does not introduce significant overhead.

Component Benchmarks The remaining cryptographic components were benchmarked separately, showing that Ed25519 signing is slower than X25519 key exchange, while AES-GCM encryption is extremely fast.

- **X25519 Key Exchange:** 1.56ms per operation.
- **Ed25519 Signing:** 4.06ms per operation.
- **Ed25519 Verification:** 4.63ms per operation.

- **AES-GCM Encryption + HMAC:** 22µs per operation.

A full breakdown of these benchmarks, including message size effects, is provided in Appendix A.

3 Implementing the SPAKE2 Protocol

3.1 Design and API Considerations

My implementation of SPAKE2 follows RFC 9382 [4] while maintaining a structured and modular design. Similar to the SIGMA implementation, I prioritized clarity and ease of use, encapsulating key functionality in dedicated classes:

- `SPAKE2Party` - Represents each participant, handling key exchange, password hashing, and transcript computation.
- `SPAKE2Handshake` - Orchestrates the handshake process, ensuring consistency and correctness.
- `SecureChannel` - Provides encrypted messaging, identical to the implementation used in SIGMA.

Unlike my X25519/Ed25519 implementation in earlier assignments, SPAKE2 does not share common utilities with other protocols. This ensures modularity, allowing it to function independently without unnecessary dependencies.

3.2 Key Derivation and Security Considerations

SPAKE2 relies on the protocol transcript `TT` to derive shared symmetric secrets:

- `TT` uniquely identifies a session and is secret to both parties, though it includes some non-secret elements.
- K_e and K_a are derived as $K_e \parallel K_a = \texttt{H}(\texttt{TT})$, where `H` is SHA-256, ensuring a minimum of 128-bit security.
- My implementation then uses a HKDF with K_a to derive the confirmation keys, K_{cA} and K_{cB} as specified in the RFC.

For password hashing, the RFC requires a memory-hard function (MHF). Initially, I used SHA-512, but this lacked memory hardness. To improve security, I switched to Argon2, significantly increasing password resistance against offline attacks. However, as shown in benchmarks, this also introduced substantial performance overhead.

3.3 SPAKE2Party: Core Functionality

`SPAKE2Party` is the primary class handling key exchange and message processing. It integrates key derivation within the class to streamline execution and reduce unnecessary external dependencies.

A challenge I faced was designing the `compute_transcript()` function. Initially, I implemented it as a global function to ensure a canonical transcript. However, this approach broke SPAKE2's security model, as both parties would derive keys from the same transcript order. To resolve this:

- I moved transcript computation into each party's class, ensuring each participant computes it independently.
- I introduced a role system where parties using `M` are designated as "A" and those using `N` as "B", ensuring a standardized transcript structure.

This approach maintains security guarantees while preventing mismatches.

3.4 SPAKE2Handshake: Managing Key Exchange

The `SPAKE2Handshake` class simplifies the interaction between parties and ensures correctness. It manages:

- Public key exchange and peer verification.
- Shared secret computation and transcript validation.
- Key derivation and confirmation message exchange.
- Error handling for mismatched keys or transcripts.

By encapsulating these steps, the handshake implementation remains clear, enforcing security checks at each stage.

4 Testing and Validation

To ensure correctness and security, I implemented a number of tests covering key aspects of SPAKE2. The tests validate password hashing, key exchange, message authentication, and secure communication.

Key test cases include:

- **Shared Secret Validation**: Ensuring both parties compute identical shared secrets.
- **Public Message Integrity**: Verifying that exchanged messages belong to the correct cryptographic group.
- **Key Schedule Consistency**: Checking that derived keys are of the correct size.
- **Handshake Confirmation**: Validating that confirmation MACs authenticate the handshake.
- **Secure Channel Functionality**: Ensuring messages are correctly encrypted and decrypted.

Additionally, edge cases were tested:

- **Password Mismatch**: Confirming failure when parties use different passwords.
- **Multiple Handshakes**: Ensuring independent secrets are generated across multiple sessions.

All tests pass successfully, verifying the implementation aligns with RFC 9382 and maintains security guarantees.

4.1 Performance and Benchmarking

I benchmarked key operations using `timeit` with warm-up runs to eliminate cache effects. The results highlight the performance impact of switching from SHA-512 to Argon2 for password hashing.

- **Handshake Time:** 20.14ms avg, 20.09ms min, 20.23ms max.
- **Key Exchange:** 5.90ms avg.
- **AES-GCM Encryption:** 22µs avg.

After switching to Argon2:

- **Handshake Time:** 298.33ms avg, 296.59ms min, 300.15ms max.
- **Key Exchange:** 5.68ms avg.
- **AES-GCM Encryption:** 22µs avg.

Metric	**SHA-512 (ms)**	**Argon2 (ms)**
Avg Handshake Time	20.14	298.33
Min Time	20.09	296.59
Max Time	20.23	300.15
P90 Latency	20.27	300.27
P99 Latency	20.32	300.42

Table 2: SPAKE2 Handshake Benchmark (SHA-512 vs. Argon2)

Analysis:

- Argon2 increased handshake time by nearly *15x*, making the protocol significantly slower.
- The SPAKE2 key exchange and AES-GCM encryption remained unaffected.
- A flame graph (see Appendix B) confirms that Argon2 dominates execution time.
- While Argon2 improves password security, the increased latency may impact usability, especially in interactive settings.

4.2 Comparison with Python's SPAKE2 Implementation

I considered benchmarking against `python-spake2`, but differences in:

- Password hashing (Argon2 vs. internal KDFs).
- Transcript computation methods.
- KDF derivation approaches.

make direct comparisons difficult. Nonetheless, the results confirm that my implementation aligns closely with RFC specifications.

4.3 Conclusion

This SPAKE2 implementation closely follows the RFC while maintaining a clear, modular API. Key takeaways include:

- A structured API improves usability and security.
- Proper transcript ordering is crucial to prevent inconsistencies.
- While Argon2 enhances password security, it introduces significant performance trade-offs.
- Overall, the implementation is robust, with all tests passing successfully.

5 Future Work and Conclusion

This project was significantly more engaging than Assignment 1, as implementing full protocols provided clearer validation—seeing both parties derive the same secret and successfully encrypt and decrypt messages gave immediate feedback on correctness. Additionally, building everything from scratch made the process rewarding and reinforced the importance of designing usable APIs. Using my own previous work for Ed25519 and X25519 also highlighted how API choices impact developer experience, emphasizing the need for clean, structured interfaces.

5.1 Potential Extensions and Optimizations

Despite the success of the implementation, there are several areas for improvement:

- **Field Multiplication Optimization**: Profiling revealed that `field_mul` is the main bottleneck in X25519 and Ed25519 operations (see Appendix C for flame graphs). While optimizing this function directly is an option, replacing my implementations with an optimized cryptographic library would significantly improve performance, but at the cost of moving away from a fully self-written implementation. I experimented with Numba [5] for JIT compilation, but it does not support the big integers used in X25519/Ed25519. I also tried [6], but surprisingly, it did not yield any noticeable performance improvements.
- **Enhancing SIGMA's Certificate Handling**: The current certificate authority (CA) implementation is minimal, and an extended version could incorporate revocation lists and validity periods, improving real-world applicability.

- **Balancing Argon2 Parameters for SPAKE2**: Switching to Argon2 for password hashing drastically increased handshake time (from 20ms to nearly 300ms). Exploring different Argon2 parameter configurations could help strike a balance between security and efficiency, making the implementation more practical for real-world use.

References

1. Aleem, F. *Assignment 1: Curve25519 Diffie-Hellman and Ed25519 Signatures* P79 Course Assignment 1 (University of Cambridge, Computer Laboratory, Feb. 2024) (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: 2025-03-01). 2025 (Page 3).
3. Krawczyk, H. *SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie Hellman and Its Use in the IKE Protocols* in *Advances in Cryptology - CRYPTO 2003* (ed Boneh, D.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2003), 400–425 (Pages 3, 5).
4. Ladd, W. *SPAKE2, a Password-Authenticated Key Exchange* RFC 9382. Sept. 2023 (Page 7).
5. Lam, S. K., Pitrou, A. & Seibert, S. *Numba: A LLVM-based Python JIT Compiler* in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (2015), 1–6 (Page 10).
6. Horsen, C. V. *gmpy2: Multiple-precision arithmetic in Python* Python package. Version 2.2.1, accessed March 3, 2025. 2025 (Page 10).

A SIGMA Benchmark Results

This appendix contains the full benchmark results for the SIGMA protocol and its cryptographic operations.

Table 3: Full Benchmark Results for SIGMA Protocol Components

Operation	**Avg Time (ms)**	**Median**	**Min**	**Max**	**P90**	**P99**
SIGMA Handshake	37.45	37.45	37.11	37.92	38.05	38.22
SIGMA-I Handshake	37.30	37.31	37.25	37.34	37.36	37.37
X25519 Key Exchange	1.56	1.56	1.55	1.56	1.57	1.57
Ed25519 Signing	4.06	4.06	4.04	4.08	4.09	4.10
Ed25519 Verification	4.63	4.63	4.62	4.65	4.66	4.67
AES-GCM (64B)	0.024	0.023	0.023	0.024	0.025	0.025
AES-GCM (1MB)	10.48	10.38	10.32	10.84	10.98	11.16

A.1 Message Size Effects on AES-GCM

To analyze the impact of message size on AES-GCM encryption performance, I tested both 64-byte and 1MB messages.

- Small messages (64B) encrypt/decrypt in 24µs.
- Large messages (1MB) take approximately 10.48ms.

This confirms that AES-GCM scales linearly with message size but remains highly efficient.

A.2 Observations and Insights

- AES-GCM encryption is extremely fast, adding negligible overhead to SIGMA-I.
- X25519 key exchange is significantly faster than Ed25519 signing. This explains why SIGMA and SIGMA-I have similar execution times.
- Ed25519 verification is slightly slower than signing, but both are computationally expensive compared to AES-GCM.
- The difference between SIGMA and SIGMA-I is within statistical variation, showing that adding identity encryption does not introduce significant latency.

B SPAKE2 Benchmark Results

This appendix presents the detailed benchmark results for the SPAKE2 implementation, including the handshake, key exchange, and secure channel performance.

B.1 SPAKE2 Handshake Benchmark

The tables below provide the full statistics for the SPAKE2 handshake timing over 100 iterations using SHA-512 (Table 4) and Argon2 (Table 5).

Metric	**Time (s)**
Average Time	0.02014
Median Time	0.02014
Standard Deviation	0.0000545
Min Time	0.02010
Max Time	0.02024
P90 Latency	0.02027
P99 Latency	0.02033

Table 4: SPAKE2 Handshake Benchmark (SHA-512)

Metric	**Time (s)**
Average Time	0.2983
Median Time	0.2983
Standard Deviation	0.00167
Min Time	0.2966
Max Time	0.3002
P90 Latency	0.3003
P99 Latency	0.3004

Table 5: SPAKE2 Handshake Benchmark (Argon2)

B.2 SPAKE2 Key Exchange Benchmark

Table 6 presents the results for SPAKE2 key exchange over 1000 iterations.

Metric	Time (s)
Average Time	0.00569
Median Time	0.00567
Standard Deviation	0.000028
Min Time	0.00566
Max Time	0.00573
P90 Latency	0.00575
P99 Latency	0.00577

Table 6: SPAKE2 Key Exchange Benchmark

B.3 Secure Channel Benchmark

The performance of AES-GCM encryption and HMAC authentication was also measured (Table 7).

Table 7: Secure Channel (AES-GCM + HMAC) Benchmark

Metric	Time (s)
Average Time	0.00002256
Median Time	0.00002233
Standard Deviation	0.00000037
Min Time	0.00002225
Max Time	0.00002298
P90 Latency	0.00002300
P99 Latency	0.00002302

B.4 Flamegraph: SPAKE2 Handshake Overhead

The following figure provides a flamegraph representation of the computational overhead during the SPAKE2 handshake. This visualization highlights the increased cost due to Argon2 password hashing.

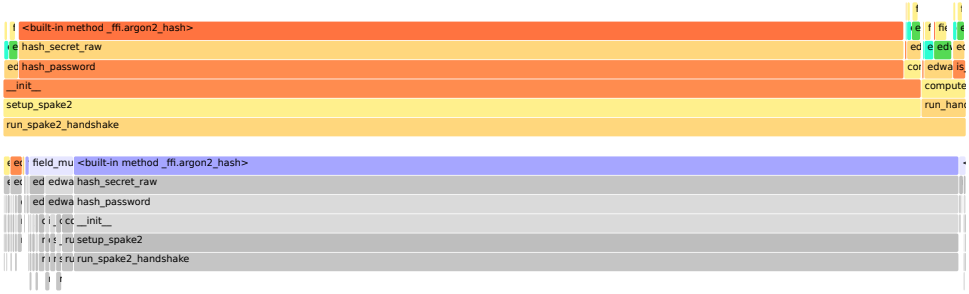

Figure 1: Flamegraph of SPAKE2 Handshake Execution (Argon2 Overhead Highlighted)

C Profiling X25519 and Ed25519 Operations

This section presents the full benchmarking and profiling results for X25519 key exchange, Ed25519 signing, and Ed25519 verification. The profiling data was used to generate flame graphs to visualize function hotspots.

C.1 Benchmark Results

Metric	Ed25519 Signing	Ed25519 Verification	X25519 Key Exchange
Avg Time	4.14	4.93	1.62
Median Time	4.14	4.91	1.61
Std Deviation	0.0168	0.0973	0.0225
Min Time	4.11	4.85	1.60
Max Time	4.16	5.10	1.65
P90 Latency	4.16	5.17	1.67
P99 Latency	4.16	5.27	1.69

Table 8: Ed25519 and X25519 Benchmark Results (Times in ms)

C.2 Flame Graphs

The following figures show flame graphs for each operation, highlighting where the most computation time is spent. As we can see in the figures, as well as in the profiling data, the `field_mul` function is the main computational bottleneck in both X25519 and Ed25519 operations, particularly due to its repeated use in scalar multiplication and point operations.

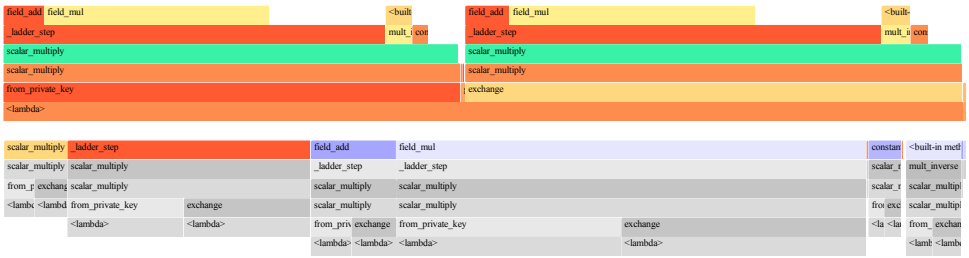

Figure 2: Flame graph for X25519 key exchange

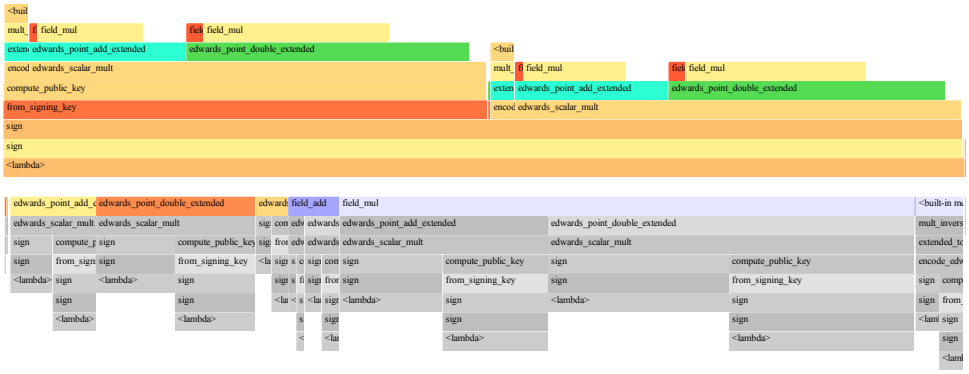

Figure 3: Flame graph for Ed25519 signing

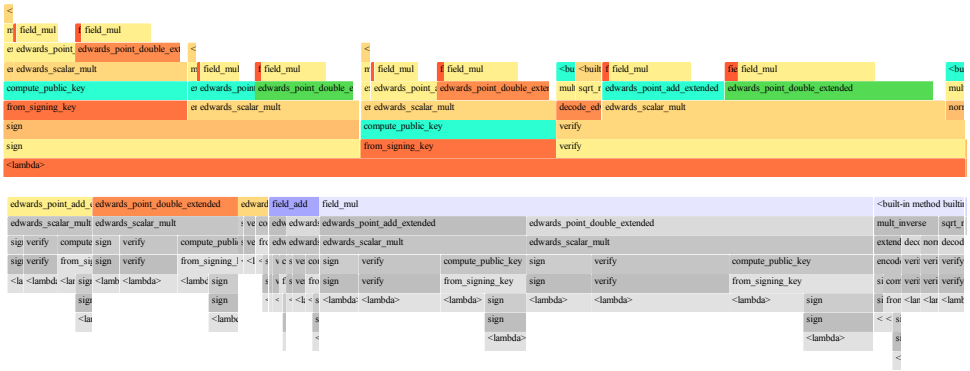

Figure 4: Flame graph for Ed25519 verification