



Security Audit Report

Polkadot micro-sr25519

v1.0

June 12, 2025

Table of Contents

Table of Contents	2
License	3
Disclaimer	4
Introduction	5
Purpose of This Report	5
Codebase Submitted for the Audit	5
Methodology	6
Functionality Overview	6
How to Read This Report	7
Code Quality Criteria	8
Summary of Findings	9
Detailed Findings	10
1. Missing validation for point at infinity	10
2. Ambiguous transcript construction due to empty label	10
3. Missing flag to enable improved transcription ordering for VRF	11
4. Incomplete signature format validation may allow non-canonical inputs	11
5. Potential timing side-channel in scalar arithmetic operations	12
6. Insufficient input validation	12
7. The chain code is generated but not returned to the caller	12
8. Insecure RNG injection in signing and VRF functions	13
9. Lack of input size restrictions may allow denial of service attacks	13
10. Misleading error message in VRF output point identity check	14
11. Presence of TODOs and pending items	14

License



THIS WORK IS LICENSED UNDER A [CREATIVE COMMONS ATTRIBUTION-NODERIVATIVES 4.0 INTERNATIONAL LICENSE](https://creativecommons.org/licenses/by-nc/4.0/).

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUCT ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

COPYRIGHT OF THIS REPORT REMAINS WITH THE AUTHOR.

This audit has been performed by

Oak Security GmbH

<https://oaksecurity.io/>
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by Edgeware DAO Association to perform a security audit of micro-sr25519.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/paulmillr/micro-sr25519
Commit	08dc56e09aab971e7fd5b2f20a6f06c11d4a8daf
Scope	All files were in scope.
Fixes verified at commit	01f903de2c79cfeb71c499d0e9538d0be8b93dc5 Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
 - a. Race condition analysis
 - b. Under-/overflow issues
 - c. Key management vulnerabilities
4. Report preparation

Functionality Overview

The `micro-sr25519` is a TypeScript implementation of the `sr25519` cryptographic scheme used in the Polkadot ecosystem.

The library provides Schnorr signature functionality on Ristretto compressed Ed25519 curves, including basic operations for key generation, message signing, and signature verification. It implements Hierarchical Deterministic Key Derivation (HDKD), supporting both hard and soft derivation methods for generating child keys from parent keys. The library also includes Verifiable Random Function (VRF) capabilities for generating cryptographically secure random outputs with proofs of correctness.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending**, **Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Low	The code is straightforward and closely resembles the reference implementation
Code readability and clarity	High	The code is readable and easy to follow
Level of documentation	Low	The code does not contain thorough documentation. Even though it is based on a reference implementation, some implementation differences are only noted as one-line code comments.
Test coverage	Medium	<p>The project contains some unit tests and uses the ZeroRNG random function to make sure test cases are reproducible. However, they could be extended to include other RNG functions, such as ChaCha20RNG, as well as fuzz tests.</p> <p>The tests in <code>test/basic.test.js:201,235</code> redundantly compare identical public keys, making the assertions trivially true.</p>

Summary of Findings

No	Description	Severity	Status
1	Missing validation for point at infinity	Minor	Acknowledged
2	Ambiguous transcript construction due to empty label	Minor	Acknowledged
3	Missing flag to enable improved transcription ordering for VRF	Minor	Acknowledged
4	Incomplete signature format validation may allow non-canonical inputs	Minor	Acknowledged
5	Potential timing side-channel in scalar arithmetic operations	Minor	Resolved
6	Insufficient input validation	Minor	Acknowledged
7	The chain code is generated but not returned to the caller	Minor	Acknowledged
8	Insecure RNG injection in signing and VRF functions	Informational	Acknowledged
9	Lack of input size restrictions may allow denial of service attacks	Informational	Acknowledged
10	Misleading error message in VRF output point identity check	Informational	Resolved
11	Presence of TODOs and pending items	Informational	Resolved

Detailed Findings

1. Missing validation for point at infinity

Severity: Minor

In `index.ts:308-311` and `index.ts:484-487`, there is no validation that the public key is the point at infinity.

The identity point may invalidate signature scheme security, since scalar multiplication by zero yields the identity, an attacker could use it as a public key to pass signature verification without a secret key.

Recommendation

We recommend adding checks that the input data is not the point at infinity.

Examples of similar validation can be found in ChainSafe's `go-schnorrkel`, specifically in `sign.go:133`, `vrf.go:271`.

Status: Acknowledged

2. Ambiguous transcript construction due to empty label

Severity: Minor

In `index.ts:200`, the `label` method in `SigningContext` invokes `appendMessage` with an empty string as the label.

In Merlin transcripts, labels are critical for domain separation and context binding.

Consequently, using an empty label can result in ambiguous or overlapping transcript states, undermining the uniqueness guarantees of the transcript.

Recommendation

We recommend always using a unique, descriptive label when absorbing context data into the transcript.

Status: Acknowledged

3. Missing flag to enable improved transcription ordering for VRF

Severity: Minor

In `index.ts:381-415`, the public key is committed to the transcript after the nonce, aligning with the Kusama ordering scheme.

This contrasts with the [secure ordering recommended in this discussion](#), where the public key commit precedes the nonce. The current ordering maintains compatibility with Polkadot and Kusama but diverges from the strategy that mitigates risks of attacks exploiting discrepancies between public and secret key alignments.

This vulnerability could be leveraged by a malicious actor to undermine the VRF's security assumptions, particularly in environments expecting stronger cryptographic assurances.

Recommendation

We recommend introducing a configurable flag in the library, analogous to the `KUSAMA_VRF` parameter in the Schnorrkel Rust implementation.

This flag should allow developers to opt into the more secure transcript ordering.

Status: Acknowledged

4. Incomplete signature format validation may allow non-canonical inputs

Severity: Minor

In `index.ts:301-304`, the signature verification logic only partially enforces the sr25519/Schnorrkel specification.

While it correctly checks for the presence of the Schnorrkel marker by verifying that the most significant bit (bit 7) of the final signature byte is set, it neglects to validate that the remaining bits in that byte, bits 0 through 6, are cleared. According to the sr25519 specification, these bits must be zero to ensure canonical signature encoding.

Failing to enforce this requirement can result in the acceptance of non-canonical signatures and could undermine the strict format guarantees that cryptographic protocols rely on for integrity and interoperability.

Recommendation

We recommend updating the verification logic to fully enforce the sr25519 specification by ensuring that only the Schnorrkel marker bit is set in the final byte of the signature and that all other bits are properly cleared.

Status: Acknowledged

5. Potential timing side-channel in scalar arithmetic operations

Severity: Minor

Scalar arithmetic operations in the codebase may be susceptible to timing side-channel attacks due to variable-time behavior in the underlying `bigint` implementation.

In JavaScript environments, the risk is mitigated to some extent by execution engine optimizations, which obscure precise timing characteristics. However, these protections are not absolute, and timing analysis remains a viable attack vector, particularly in high-value or adversarial settings.

Recommendation

We recommend considering explicit constant-time implementations for critical paths.

Status: Resolved

6. Insufficient input validation

Severity: Minor

The `secretFromSeed` and `getSharedSecret` functions, defined respectively in `index.ts:247` and `index.ts:320`, lack comprehensive input validation.

For example, they do not check for zeroed arrays or structurally invalid inputs, which could lead to incorrect computations.

Recommendation

We recommend implementing robust input validation for these functions, including checks for zeroed data and correct format and length.

Status: Acknowledged

7. The chain code is generated but not returned to the caller

Severity: Minor

In `index.ts:363,373`, the execution calculates chain code using a `SigningContext`, specifically by calling the `challengeBytes` method.

Despite calculating bytes intended as a new chain code, these bytes are discarded and not returned by the function.

Consequently, this is inefficient and fails to leverage the dynamically generated chain code for further key derivation or processing.

Recommendation

We recommend returning the generated chain code as in the Schnorrkel Rust implementation.

Status: Acknowledged

8. Insecure RNG injection in signing and VRF functions

Severity: Informational

The signing and verifiable random function (VRF) routines in the library accept an overridable `rng` parameter. This introduces a security risk if the supplied RNG is weak, deterministic, or replayable. Under such conditions, the nonce values used in cryptographic operations become predictable, enabling a malicious actor to derive private keys.

This risk is exacerbated by the use of `randomBytes` from `@noble/hashes/utils`, which may default to insecure sources in environments lacking a cryptographically secure pseudo-random number generator (CSPRNG). Such misconfiguration could occur due to compromised dependencies or developer oversight, creating exploitable conditions for key leakage.

Recommendation

We recommend implementing one or more of the following mitigations:

- Enforce usage of a secure, internal CSPRNG without accepting external RNG parameters.
- If parameterization is necessary, document the security assumptions clearly and rename the functions to indicate the reliance on external RNG input.

Status: Acknowledged

9. Lack of input size restrictions may allow denial of service attacks

Severity: Informational

The audited library does not enforce maximum length constraints on input parameters, including `message`, `context`, and `extra`.

In environments where this library is leveraged in a backend service, unrestricted input sizes pose a denial of service (DoS) risk.

An attacker could exploit this by submitting excessively large inputs, which would result in severe performance degradation due to the library's internal byte-by-byte processing using 166-byte chunks (`STROBE_R`). Operations like `absorb`, `squeeze`, and `overwrite`, and VRF functionalities in `index.ts:453-454` and `index.ts:478-479` are particularly susceptible.

Additionally, constructs like `SigningContext` in JavaScript environments allow allocation of `Uint8Array` instances exceeding 4GB, exacerbating the potential impact of the lack of input size restrictions.

Recommendation

We recommend documenting the performance implications of large input sizes and annotating the affected functions with clear code comments.

Specifically, implementors should be advised to apply strict input size validation in the `sign`, `verify`, `vrf.sign`, `vrf.verify` operations.

Status: Acknowledged

10. Misleading error message in VRF output point identity check

Severity: Informational

In `index.ts: 493-496`, the VRF verification function includes a check to detect the identity (zero) point as the output.

While this validation is correctly implemented, the associated error message inaccurately suggests that the identity check applies to the public key rather than the output point. This misrepresentation may confuse developers and hinder debugging or security assessments.

Recommendation

We recommend updating the error message to accurately reflect that the identity point check pertains to the VRF output point, not the public key.

Status: Resolved

11. Presence of TODOs and pending items

Severity: Informational

The audited codebase includes unresolved TODO comments and pending items, which represent incomplete or unverified segments of the code.

It is best practice to resolve them before the code is released into production.

Specific instances were identified at:

- `index.ts:73`
- `index.ts:260`

Recommendation

We recommend removing or resolving all TODO comments and pending items prior to release.

Status: Resolved



Differential Fuzz Testing Report

Polkadot micro-sr25519

v1.0

June 12, 2025

Table of Contents

Table of Contents	2
License	3
Disclaimer	4
Introduction	5
Purpose of This Report	5
Codebase Submitted for the Audit	6
Methodology	7
Functionality Overview	7
Differential fuzzing methodology and results	8
Overview	8
Architecture	8
Observations	9
Tested operations	9
How to Read This Report	11
Code Quality Criteria	12
Summary of Findings	13
Detailed Findings	14
1. micro-sr25519 secret keys are encoded as ed25519 bytes, which is different than schnorrkel default encoding	14
Appendix A: Test Cases	15
1. Test case for “micro-sr25519 secret keys are encoded as ed25519 bytes, which is different than schnorrkel default encoding”	15

License



THIS WORK IS LICENSED UNDER A [CREATIVE COMMONS ATTRIBUTION-NODERIVATIVES 4.0 INTERNATIONAL LICENSE](https://creativecommons.org/licenses/by-nc/4.0/).

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUCT ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

COPYRIGHT OF THIS REPORT REMAINS WITH THE AUTHOR.

This audit has been performed by

Oak Security GmbH

<https://oaksecurity.io/>
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by Edgeware DAO Association to perform a security audit of micro-sr25519.

This report concerns the differential fuzz testing of the TypeScript micro-sr25519 implementation (paulmillr/micro-sr25519) against the Rust schnorrkel reference implementation (w3f/schnorrkel). The objective of this effort is to discover inconsistencies between the two implementations by means of differential fuzzing and to report any issues or unexpected behavior.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/paulmillr/micro-sr25519
Commit	08dc56e09aab971e7fd5b2f20a6f06c11d4a8daf
Scope	All files were in scope.

Methodology

The differential fuzz testing was conducted in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Execute unit tests for both the JavaScript and Rust implementations to establish baseline correctness.
3. Enumerate fuzz targets for key operations and run `cargo fuzz run` for each target in parallel.
4. Set up a Dockerfile to enable easy reproduction of the fuzz tests in containerized environments.
5. Run the differential fuzzing setup on the Hetzner's cpx51 cloud server (16 vCPU EPYC 7002, 32GB RAM) for 48h.
6. Monitor and log discrepancies between the JavaScript and Rust implementations, capturing any diverging cases.
7. Analyze and classify any issues discovered, then prepare the final report.

Functionality Overview

The `micro-sr25519` is a TypeScript implementation of the `sr25519` cryptographic scheme used in the Polkadot ecosystem.

The library provides Schnorr signature functionality on Ristretto compressed Ed25519 curves, including basic operations for key generation, message signing, and signature verification. It implements Hierarchical Deterministic Key Derivation (HDKD), supporting both hard and soft derivation methods for generating child keys from parent keys. The library also includes Verifiable Random Function (VRF) capabilities for generating cryptographically secure random outputs with proofs of correctness.

Differential fuzzing methodology and results

Overview

The differential fuzzing suite implements a fuzz testing harness for the micro-sr25519 TypeScript library.

The suite aims to identify mismatches in signing, verification, key derivation (HDKD), and verifiable random function (VRF) outputs through systematic input generation and instrumentation.

Instrumentation and campaign logic are available at: <https://github.com/oak-security/polkadot-micro-sr25519-fuzz>.

Architecture

To minimize performance overhead, a persistent Node.js subprocess is spawned from a Rust orchestrator. This avoids reinitializing the V8 engine for each test case.

Communication between the Rust orchestrator and the Node.js runtime is performed over stdin/stdout, using a lightweight line-delimited JSON protocol for structured messages.

The fuzzer is built in Rust, using `cargo-fuzz` and `libFuzzer` to perform coverage-guided mutations of inputs.

The Rust Schnorrkel library is used as an oracle to deterministically generate correct signatures, keys, and VRF outputs. Inputs are mutated by `libFuzzer` to maximize code coverage in the TypeScript implementation. Malformed input is not tested by the differential fuzz targets, due to the time constraints of the time-boxed security review.

Each operation is developed as a separate fuzz target, which receives a stream of random u8 bytes (ranging from 0 to 4096 bytes by default in `libFuzzer`) and extracts the appropriate variables necessary for the function inputs. For example, the `sign` target uses the first 32 bytes for the seed, the next byte for the RNG function definition, and the remaining bytes for the message to be signed. It then signs the message using Rust's library, sends the same input to the Node.js subprocess, and compares the response outputs.

Three different RNG implementations, Zero, Incrementer, and ChaCha20, are supported to test constant, incremental, and pseudo-random input patterns, respectively. As the report shows, this was paramount to uncover Issue 1 highlighted in this report, which is not apparent by using only a constant random number generator function (Zero RNG).

A continuous integration script was included in the differential fuzzing repository to reuse the corpus on new pushes and can later be integrated into the main micr-sr25519 repository to automatically check for regressions on future updates.

Observations

The 4 KiB input size limit proved sufficient for effective fuzzing, as all tested primitives consume 128 bytes or less, and increasing the buffer size to 16 KiB had no measurable impact on basic block coverage. While larger inputs may influence deeper hash or codec branches, they offered diminishing returns in this context.


An entropic scheduling strategy was employed, but quickly reached a plateau, as it could be seen in Jazzer logs. The final corpus was trimmed down to just three inputs without any loss in coverage.

Throughout the campaign, the system remained stable: no hangs or out-of-memory conditions were observed, and memory usage consistently stayed under 900 MiB.

Tested operations

The operations were tested on Hetzner's cpx51 cloud server (16 vCPU EPYC 7002, 32GB RAM) for 48h.

Operation	Fuzz target	Passing
<code>sr25519.sign(pair.secretKey, msg)</code>	sign	✓
<code>sr25519.verify(msg, polkaSig, pair.publicKey)</code>	verify	✓
<code>sr25519.secretFromSeed(seed)</code>	secret_from_seed	✓
<code>sr25519.getPublicKey(secretKey)</code>	get_public_key	✓
<code>sr25519.getSharedSecret(secretKey, publicKey)</code>	get_shared_secret	✓
<code>sr25519.HDKD.secretHard(pair.secretKey, cc)</code>	secret_hard	✓
<code>sr25519.HDKD.secretSoft(pair.secretKey, cc)</code>	secret_soft	✓
<code>sr25519.HDKD.publicSoft(pubSelf, cc)</code>	public_soft	✓
<code>sr25519.vrf.sign(msg, pair.secretKey)</code>	vrf_sign	✓

<code>sr25519.vrf.verify(msg, sig, pair.publicKey)</code>	<code>vrf_verify</code>	
---	-------------------------	---

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending**, **Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Low	The code is straightforward and closely resembles the reference implementation
Code readability and clarity	High	The code is readable and easy to follow
Level of documentation	Low	The code does not contain thorough documentation. Even though it is based on a reference implementation, some implementation differences are only noted as one-line code comments.
Test coverage	Medium	The project contains some unit tests and uses the ZeroRNG random function to make sure test cases are reproducible. However, they could be extended to include other RNG functions, such as ChaCha20RNG, as well as fuzz tests.

Summary of Findings

No	Description	Severity	Status
1	micro-sr25519 secret keys are encoded as ed25519 bytes, which is different than schnorrkel default encoding	Informational	Acknowledged

Detailed Findings

1. **micro-sr25519 secret keys are encoded as ed25519 bytes, which is different than schnorrkel default encoding**

Severity: Informational

In `index.ts:255,266,350,368`, the `micro-sr25519` TypeScript functions encode secret keys using the ed25519 byte format, whereas the Rust `schnorrkel` reference uses a different internal format by default.

More specifically, calling `keypair.secret.to_bytes` from `schnorrkel` yields a different output than `sr25519.secretFromSeed`. To have the same output, `to_ed25519_bytes` should be used.

This inconsistency can lead to interoperability issues if keys are shared directly between the implementations.

A test case showcasing the issue is provided in the [Appendix](#).

Recommendation

We recommend documenting the encoding difference in the `micro-sr25519` README and providing helper functions to convert between the ed25519-based format and the `schnorrkel` format.

Status: Acknowledged

Appendix A: Test Cases

1. Test case for “[micro-sr25519 secret keys are encoded as ed25519 bytes, which is different than schnorrkel default encoding](#)”

```
fn get_test_keypair_from_seed(seed_hex: &str) -> Keypair {
    let seed_bytes = hex::decode(seed_hex).unwrap();
    let seed: [u8; 32] = seed_bytes.try_into().unwrap();
    let mini = MiniSecretKey::from_bytes(&seed).unwrap();
    mini.expand_to_keypair(ExpansionMode::Ed25519)
}

#[test]
fn test_secret_from_seed() {
    let seed_hex =
        "0afffffffffffffffffffffffffffffffffffffffffff05000000000000";
    let kp = get_test_keypair_from_seed(seed_hex);
    let secret_key_ed25519_bytes = kp.secret.to_ed25519_bytes();
    assert_eq!(hex::encode(secret_key_ed25519_bytes),

        "487908c2cf7893dbaf0a658031d97553724c277d8094e4327091f98139398153c48de404fc3c07d
        5ade70ee7730e34aad5ca8a2dedc85b618a19b3a2a408f9f0"
    );
}
```

```
function getTestKeypairFromSeed(seedString) {
    const seed = new Uint8Array(Buffer.from(seedString, "hex"));
    const secretKey = sr25519.secretFromSeed(seed);
    const publicKey = sr25519.getPublicKey(secretKey);
    return { secretKey, publicKey };
}

test("test_secret_from_seed", () => {
    const { secretKey } = getTestKeypairFromSeed(
        "0afffffffffffffffffffffffffffffffffffffffffff05000000000000",
    );
    expect(Buffer.from(secretKey).toString("hex")).toBe(
        "487908c2cf7893dbaf0a658031d97553724c277d8094e4327091f98139398153c48de404fc3c07d
        5ade70ee7730e34aad5ca8a2dedc85b618a19b3a2a408f9f0",
    );
});
```