



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",
    );
});
```