## Security Audit Report

# Polkadot scure-sr25519

**v1.6**

**August 22, 2025**

# Table of Contents

# License

# 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 CONSTRUE 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.



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 scure-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/scure-sr25519 |
| Commit | Initial audit:<br>`08dc56e09aab971e7fd5b2f20a6f06c11d4a8daf`<br><br>Follow-up review focused on cryptographic components:<br>`a4a538b18ffecebacdacc9e862adc47938b9a82d` |
| Scope | All files were in scope. |
| Fixes verified at commit | `6f0ed8bc9123a6ed81cc349cefd90d8be16b90e1` |

| | |
|---|---|
| | 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
5. Following the initial audit and report preparation, a focused review of the cryptographic components was conducted by one additional security researcher, which included:
    a. Regression analysis to verify the status of previously reported findings.
    b. Isolated analysis of the cryptographic components `Merlin`, `STROBE`, `DLEQ`, and `VRF` to ensure their individual implementations were sound.
    c. The implementation of Schnorr signatures over `Ristretto/Curve25519` was analyzed in detail, verifying the correctness of mathematical operations and adherence to protocol specifications.

# Functionality Overview

The scure-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 | 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.<br><br>The tests in `test/basic.test.js:201` and `235` redundantly compare identical public keys, making the assertions trivially true. |

# 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 | Resolved |
| 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: Resolved**

## 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**