@anvil-vault/csl

Cardano Serialization Library (CSL) wrappers and utilities for Anvil Vault. This package provides type-safe, Result-based wrappers around the Emurgo Cardano Serialization Library, making it easier to work with Cardano cryptographic operations.

Installation

```
npm install @anvil-vault/csl
```

Overview

The CSL package provides:

- **Key Derivation**: BIP32 hierarchical deterministic key derivation
- **Address Generation**: Create base, enterprise, and reward addresses
- **Transaction Signing**: Sign transactions with private keys
- **Data Signing**: Sign arbitrary data with Ed25519 keys
- **Signature Verification**: Verify Ed25519 signatures
- **Address Parsing**: Parse addresses from various formats
- **Network Utilities**: Network ID and configuration helpers

All functions return `Result` types from the `trynot` library for consistent error handling.

API Reference

Key Derivation

`deriveAccount(input)`

Derives an account key from a root key following CIP-1852 (Cardano's hierarchical deterministic wallet structure).

Parameters:

- `input.rootKey: Bip32PrivateKey | string` - Root private key (BIP32 extended key)
- `input.accountDerivation: number | number[]` - Account index or derivation path

Returns: `Result<DeriveAccountOutput>`

- `rootKey: Bip32PrivateKey` - The parsed root key
- `accountKey: Bip32PrivateKey` - The derived account key

Derivation Path: `m/1852'/1815'/account'`

- `1852'` - Purpose (CIP-1852)
- `1815'` - Coin type (Cardano)

- `account'` - Account index (hardened)

Example:

```
import { deriveAccount } from "@anvil-vault/csl";
import { isErr } from "trynot";

const rootKeyHex = "40d0f8821976d097ad6c22e75f3ee2e725750a33...";

// Derive account 0
const result = deriveAccount({
  rootKey: rootKeyHex,
  accountDerivation: 0
});

if (!isErr(result)) {
  console.log("Account key derived:", result.accountKey.to_bech32());
}

// Derive with custom path
const customResult = deriveAccount({
  rootKey: rootKeyHex,
  accountDerivation: [0, 1] // m/1852'/1815'/0'/1'
});
```

`derivePrivateKey(input)`

Derives a BIP32 private key from BIP39 entropy (mnemonic seed).

Parameters:

- `input.entropy: Buffer | string` - BIP39 entropy (hex string or Buffer)
- `input.password?: Buffer | string` - Optional password for entropy (default: empty)

Returns: `Result<Bip32PrivateKey>`

Example:

```
import { derivePrivateKey } from "@anvil-vault/csl";
import { isErr } from "trynot";

const entropy = "a1b2c3d4e5f6...";
const result = derivePrivateKey({ entropy });

if (!isErr(result)) {
  console.log("Root key:", result.to_bech32());
}

// With password
const protectedResult = derivePrivateKey({
```

```
  entropy,
  password: "my-secure-password"
});
```

`extractKeys(input)`

Extracts payment and stake keys from an account key following CIP-1852.

Parameters:

- `input.accountKey: Bip32PrivateKey | string` - Account key
- `input.paymentDerivation: number | number[]` - Payment key derivation path
- `input.stakeDerivation: number | number[]` - Stake key derivation path

Returns: `Result<ExtractKeysOutput>`

- `accountKey: Bip32PrivateKey` - The parsed account key
- `paymentKey: Bip32PrivateKey` - Derived payment key (external chain)
- `stakeKey: Bip32PrivateKey` - Derived stake key (staking chain)

Derivation Paths:

- Payment: `account/0/payment` (external chain)
- Stake: `account/2/stake` (staking chain)

Example:

```
import { extractKeys } from "@anvil-vault/csl";
import { isErr } from "trynot";

const result = extractKeys({
  accountKey: accountKeyHex,
  paymentDerivation: 0,  // First payment key
  stakeDerivation: 0     // First stake key
});

if (!isErr(result)) {
  console.log("Payment key:", result.paymentKey.to_public().to_bech32());
  console.log("Stake key:", result.stakeKey.to_public().to_bech32());
}

// Multiple derivation levels
const customResult = extractKeys({
  accountKey: accountKeyHex,
  paymentDerivation: [0, 5],  // account/0/0/5
  stakeDerivation: [0]        // account/2/0
});
```

`harden(num)`

Converts a derivation index to its hardened equivalent by adding 2^31.

Parameters:

- `num: number` - Derivation index

Returns: `number` - Hardened index (num + 0x80000000)

Example:

```
import { harden } from "@anvil-vault/csl";

console.log(harden(0));    // 2147483648 (0x80000000)
console.log(harden(1852)); // 2147485500 (0x8000073C)

// Use in manual derivation
const accountKey = rootKey
  .derive(harden(1852))  // Purpose
  .derive(harden(1815))  // Coin type
  .derive(harden(0));    // Account
```

Address Generation

`deriveAddresses(input)`

Derives Cardano addresses from payment and stake keys.

Parameters:

- `input.paymentKey: Bip32PrivateKey | string` - Payment private key
- `input.stakeKey: Bip32PrivateKey | string` - Stake private key
- `input.network: Network | NetworkId` - Network ("mainnet", "preprod", "preview", or 0/1)

Returns: `Result<DeriveAddressesOutput>`

- `paymentKey: Bip32PrivateKey` - Parsed payment key
- `stakeKey: Bip32PrivateKey` - Parsed stake key
- `baseAddress: BaseAddress` - Base address (payment + stake)
- `enterpriseAddress: EnterpriseAddress` - Enterprise address (payment only)
- `rewardAddress: RewardAddress` - Reward address (stake only)

Example:

```
import { deriveAddresses } from "@anvil-vault/csl";
import { isErr } from "trynot";

const result = deriveAddresses({
  paymentKey: paymentKeyHex,
  stakeKey: stakeKeyHex,
  network: "preprod"
```

```
});

if (!isErr(result)) {
  console.log("Base:", result.baseAddress.to_address().to_bech32());
  console.log("Enterprise:",
result.enterpriseAddress.to_address().to_bech32());
  console.log("Reward:", result.rewardAddress.to_address().to_bech32());
}
```

Address Types:

- **Base Address**: Contains both payment and staking credentials. Used for receiving funds and staking.
- **Enterprise Address**: Contains only payment credential. Cannot participate in staking.
- **Reward Address**: Contains only staking credential. Used for receiving staking rewards.

`parseAddress(input)`

Parses a Cardano address from various formats (bech32, hex, or CSL object).

Parameters:

- `input.address: Address | string | ParsedAddress` - Address to parse

Returns: `Result<ParsedAddress>`

- Returns one of: `BaseAddress`, `EnterpriseAddress`, `PointerAddress`, or `RewardAddress`

Example:

```
import { parseAddress } from "@anvil-vault/csl";
import { BaseAddress } from "@emurgo/cardano-serialization-lib-nodejs";
import { isErr } from "trynot";

// Parse bech32 address
const result1 = parseAddress({
  address: "addr_test1qz..."
});

if (!isErr(result1) && result1 instanceof BaseAddress) {
  console.log("Payment credential:",
result1.payment_cred().to_keyhash()?.to_hex());
  console.log("Stake credential:",
result1.stake_cred().to_keyhash()?.to_hex());
}

// Parse hex address
const result2 = parseAddress({
  address: "00a1b2c3..."
});

// Already parsed - returns as-is
```

```
const baseAddr = BaseAddress.new(/* ... */);
const result3 = parseAddress({ address: baseAddr });
```

Transaction Operations

`signTransaction(input)`

Signs a Cardano transaction with one or more private keys.

Parameters:

- `input.transaction: Transaction | FixedTransaction | string` - Transaction to sign
- `input.privateKeys: Array<PrivateKey | string>` - Private keys for signing

Returns: `Result<SignTransactionOutput>`

- `signedTransaction: FixedTransaction` - Signed transaction
- `witnessSet: TransactionWitnessSet` - Witness set with signatures

Example:

```
import { signTransaction } from "@anvil-vault/csl";
import { isErr } from "trynot";

const txHex =
"84a500d90102818258203b1663796602c0d84b03c0f201c4ed3a76667...";

const result = signTransaction({
  transaction: txHex,
  privateKeys: [
    paymentPrivateKeyHex,
    stakePrivateKeyHex
  ]
});

if (!isErr(result)) {
  console.log("Signed TX:", result.signedTransaction.to_hex());
  console.log("Witness set:", result.witnessSet.to_hex());

  // Submit to blockchain
  await submitTransaction(result.signedTransaction.to_hex());
}
```

`addRequiredSigner(input)`

Adds a required signer key hash to a transaction.

Parameters:

- `input.transaction: Transaction | FixedTransaction | string` - Transaction to modify

- `input.keyHash: Ed25519KeyHash | string` - Key hash to add as required signer

Returns: `Result<Transaction | FixedTransaction>`

- Returns the same type as input transaction

Example:

```
import { addRequiredSigner, parseAddress } from "@anvil-vault/csl";
import { BaseAddress } from "@emurgo/cardano-serialization-lib-nodejs";
import { isErr, unwrap } from "trynot";

// Simple approach: Get key hash from wallet address
// First, get the wallet from the API
const walletResponse = await
fetch("http://localhost:3000/users/me/wallet");
const wallet = await walletResponse.json();

// Parse the base address to extract the payment key hash
const address = unwrap(parseAddress({ address:
wallet.addresses.base.bech32 }));
if (address instanceof BaseAddress) {
  const keyHash = address.payment_cred().to_keyhash()?.to_hex();

  const result = addRequiredSigner({
    transaction: txHex,
    keyHash
  });

  if (!isErr(result)) {
    console.log("Updated transaction:", result.to_hex());
  }
}

// Alternative: If you have the payment key directly
import { extractKeys } from "@anvil-vault/csl";

const { paymentKey } = unwrap(extractKeys({
  accountKey,
  paymentDerivation: 0,
  stakeDerivation: 0
}));

const keyHashFromKey =
paymentKey.to_public().to_raw_key().hash().to_hex();
const result2 = addRequiredSigner({
  transaction: txHex,
  keyHash: keyHashFromKey
});
```

Use Cases:

- Multi-signature transactions
- Smart contract interactions requiring specific signers
- Governance voting

Data Signing & Verification

`signDataRaw(input)`

Signs arbitrary data with an Ed25519 private key.

Parameters:

- `input.data: Buffer | string` - Data to sign (hex string or Buffer)
- `input.privateKey: PrivateKey | string` - Private key for signing

Returns: `Result<SignDataRawOutput>`

- `signature: Ed25519Signature` - The signature

Example:

```
import { signDataRaw } from "@anvil-vault/csl";
import { isErr } from "trynot";

const message = Buffer.from("Hello, Cardano!", "utf8");

const result = signDataRaw({
  data: message,
  privateKey: privateKeyHex
});

if (!isErr(result)) {
  console.log("Signature:", result.signature.to_hex());
}
```

`verifySignature(input)`

Verifies an Ed25519 signature against data and public key.

Parameters:

- `input.signature: Ed25519Signature | string` - Signature to verify
- `input.publicKey: PublicKey | string` - Public key
- `input.data: Buffer | string` - Original data that was signed

Returns: `Result<VerifySignatureOutput>`

- `isValid: boolean` - Whether the signature is valid

Example:

```
import { verifySignature } from "@anvil-vault/csl";
import { isErr } from "trynot";

const result = verifySignature({
  signature: signatureHex,
  publicKey: publicKeyHex,
  data: Buffer.from("Hello, Cardano!", "utf8")
});

if (!isErr(result)) {
  if (result.isValid) {
    console.log("✓ Signature is valid");
  } else {
    console.log("✗ Signature is invalid");
  }
}
```

Key Generation

`generateEd25519KeyPair()`

Generates a new random Ed25519 key pair.

Returns: `Result<GenerateKeyPairOutput>`

- `privateKey: PrivateKey` - Generated private key
- `publicKey: PublicKey` - Corresponding public key

Example:

```
import { generateEd25519KeyPair } from "@anvil-vault/csl";
import { isErr } from "trynot";

const result = generateEd25519KeyPair();

if (!isErr(result)) {
  console.log("Private key:", result.privateKey.to_hex());
  console.log("Public key:", result.publicKey.to_hex());

  // Use for signing
  const signature = result.privateKey.sign(Buffer.from("data"));
  const isValid = result.publicKey.verify(Buffer.from("data"), signature);
}
```

Use Cases:

- Generating ephemeral keys
- Testing and development
- Creating non-hierarchical keys

Network Utilities

`getNetworkId(network)`

Converts network name to network ID.

Parameters:

- `network: Network | NetworkId` - Network name or ID

Returns: `NetworkId` (0 for testnet, 1 for mainnet)

Example:

```
import { getNetworkId } from "@anvil-vault/csl";

getNetworkId("mainnet");  // 1
getNetworkId("preprod");  // 0
getNetworkId("preview");  // 0
getNetworkId(0);          // 0 (pass-through)
getNetworkId(1);          // 1 (pass-through)
```

`networks`

Array of supported network names.

Type: `readonly ["mainnet", "preprod", "preview"]`

Example:

```
import { networks } from "@anvil-vault/csl";

for (const network of networks) {
  console.log(network);
}
// Output: mainnet, preprod, preview
```

Type Definitions

Network Types

```
type Network = "mainnet" | "preprod" | "preview";
type NetworkId = number; // 0 for testnet, 1 for mainnet
```

Transaction Types

```
type TransactionInput = Transaction | FixedTransaction | string;
```

Address Types

```
type ParsedAddress = BaseAddress | EnterpriseAddress | PointerAddress |
RewardAddress;
```

Complete Example: Wallet Creation

Here's a complete example showing how to create a wallet from a mnemonic:

```
import {
  derivePrivateKey,
  deriveAccount,
  extractKeys,
  deriveAddresses
} from "@anvil-vault/csl";
import { isErr, unwrap } from "trynot";

// 1. Convert mnemonic to entropy (use @anvil-vault/bip39)
const entropy = "a1b2c3d4e5f6...";

// 2. Derive root key from entropy
const rootKey = unwrap(derivePrivateKey({ entropy }));

// 3. Derive account key (account 0)
const { accountKey } = unwrap(deriveAccount({
  rootKey,
  accountDerivation: 0
}));

// 4. Extract payment and stake keys (address 0)
const { paymentKey, stakeKey } = unwrap(extractKeys({
  accountKey,
  paymentDerivation: 0,
  stakeDerivation: 0
}));

// 5. Generate addresses
const addresses = unwrap(deriveAddresses({
  paymentKey,
  stakeKey,
  network: "mainnet"
}));

console.log("Base address:",
addresses.baseAddress.to_address().to_bech32());
console.log("Enterprise address:",
```

```
addresses.enterpriseAddress.to_address().to_bech32());
console.log("Reward address:",
addresses.rewardAddress.to_address().to_bech32());
```

Error Handling

All functions return `Result` types from the `trynot` library:

```
import { isErr, isOk, unwrap } from "trynot";
import { deriveAccount } from "@anvil-vault/csl";

// Check for errors
const result = deriveAccount({ rootKey, accountDerivation: 0 });
if (isErr(result)) {
  console.error("Failed to derive account:", result.message);
  return;
}

// Use the value
console.log("Account key:", result.accountKey.to_bech32());

// Or unwrap (throws on error)
const { accountKey } = unwrap(deriveAccount({ rootKey, accountDerivation:
0 }));
```

CIP Standards

This package follows Cardano Improvement Proposals:

- **CIP-1852**: HD Wallets for Cardano
  - Derivation path: `m/1852'/1815'/account'/chain/index`
  - Purpose: 1852' (hardened)
  - Coin type: 1815' (Cardano, hardened)
  - Account: 0' to 2^31-1 (hardened)
  - Chain: 0 (external/payment), 1 (internal/change), 2 (staking)
  - Index: 0 to 2^31-1 (non-hardened)

Dependencies

- `@emurgo/cardano-serialization-lib-nodejs-gc`: Core Cardano serialization library
- `@anvil-vault/utils`: Shared utilities (parseFromHex, error handling)
- `trynot`: Result type for error handling