ChatGPT

Implementación de derivación de direcciones HD en Cardano y Solana (Rust)

Introducción

Derivar direcciones de **Cardano (ADA)** y **Solana (SOL)** a partir de una frase semilla (*mnemonic*) requiere seguir estándares específicos de monedero determinístico (HD) adaptados a cada red. Ambos utilizan **curvas Ed25519** para las llaves, pero aplican **variantes de BIP32/BIP44** particulares (con diferentes **derivation paths** y esquemas de derivación). A continuación, se detalla en profundidad cómo se realiza la derivación desde la frase semilla hasta la dirección final en cada caso, incluyendo las curvas, estándares BIP relevantes, esquemas de firma y codificación de direcciones, con un enfoque en su implementación en **Rust**.

Cardano – Derivación HD desde frase semilla (CIP-1852)

Cardano emplea un esquema HD basado en BIP44 pero adaptado para Ed25519. En la era **Shelley** (a partir de 2020), todos los monederos Cardano utilizan el estándar **CIP-1852** (propósito `1852'` en la ruta de derivación) [1] [2]. Esto reemplaza al propósito `44'` usado en la era Byron (2017–2020), ya que en Byron se utilizaba un algoritmo distinto y no compatible para generar direcciones a partir de las llaves públicas [1] [3]. El nuevo propósito distingue claramente los monederos Shelley de los legacy Byron, evitando confusiones en la derivación [1].

Mnemonic y seed: Cardano emplea frases mnemónicas BIP-39 estándar (generalmente de 12, 15, 24 palabras) que se convierten a una **seed** de 512 bits siguiendo BIP-39 [4]. A partir de esta seed, se genera la **llave maestra** usando un **HMAC-SHA512** adaptado para Ed25519. En lugar de la cadena `"Bitcoin seed"`, Cardano utiliza la etiqueta `"ed25519 cardano seed"` (según la propuesta SLIP-0023) para este HMAC, produciendo 64 bytes iniciales [5]. Dicha salida se separa en dos partes de 32 bytes: `kL` y `kR`. Luego se **"clampean" los bits de** `kL` (ajustes según Ed25519, limpiando ciertos bits bajos y altos) para obtener una clave escalar válida [6] [7]. Estos 64 bytes (`kL || kR`) forman la **clave privada extendida maestra** (XPrv), y además se obtiene un **chain code** de 32 bytes para derivación determinística segura [6]. (Internamente, Cardano trata estos 96 bytes como: 32 bytes de clave privada escalar, 32 bytes de "IV" o datos extra usados en firmas Ed25519, y 32 bytes de chain code [6]). Este proceso sigue la especificación **Ed25519-BIP32**, la cual adapta BIP32 para curvas Edwards [8]. *Nota:* Cardano **no usa BIP32 secp256k1** estándar, sino una variante "BIP32-Ed25519" [8]. Esto es necesario porque Ed25519 genera la llave pública a través de una función hash y ajustes de bits, lo que impide usar directamente BIP32 sin adaptaciones; la solución es usar la forma extendida de Ed25519 (posterior al hashing) para permitir derivaciones HD [9].

Derivation path (ruta HD): CIP-1852 define la ruta `m/1852'/1815'/account'/role/index` para monederos Shelley [2]. Aquí: - `1852'` es el **purpose** (indicando esquema HD Shelley de Cardano) [2]. - `1815'` es el **coin type** para ADA según SLIP-44 (1815 corresponde al número de moneda de Cardano) [2]. - `account'` es el índice de cuenta (hardened). Por ejemplo `0'` para la primera cuenta. - `role` es el tipo de llave: `0` para **claves externas (recepción)**, `1` para **claves internas (cambio)**, y `2` para **claves de staking** (delegación), según CIP-1852 [10]. *(Roles adicionales* `3`, `4`, `5` *están reservados para DRep y keys de comité en propuestas de gobierno, ver CIP-0105)* - `index` es el índice de

dirección dentro de esa cuenta y rol (normalmente una derivación **no hardened**, permitiendo generar muchas direcciones públicas a partir de la llave pública extendida de la cuenta).

Un ejemplo concreto de la primera dirección de pago sería `m/1852'/1815'/0'/0/0` [11], que corresponde a la cuenta 0, cadena externa (recepción) e índice 0. La primera **llave de staking** asociada estaría en `m/1852'/1815'/0'/2/0` (rol 2, índice 0).

Derivación de hijos: Cardano implementa una **derivación híbrida**: los tres primeros niveles (purpose, coin type, account) siempre son **hardened** (indicados con `'`), mientras que las derivaciones posteriores (role, index) son por defecto **no hardened** [2]. Esto significa que a nivel de cuenta se puede obtener una **extended public key** (XPub) y derivar de ella todas las direcciones (roles 0/1) sin conocer la clave privada, útil para monederos ligeros o hardware wallets [12] [13]. La razón detrás es la misma filosofía BIP-44: con la pubkey extendida de la cuenta se pueden generar determinísticamente las direcciones públicas (soft derivation), pero dado que Ed25519 no soporta derivación pública estándar, Cardano se apoya en la variante Ed25519-BIP32 antes mencionada que sí lo permite [14]. En la práctica, Cardano usa la **implementación Icarus (HD** secuencial**)** en Shelley para derivación, que corresponde al esquema ED25519-BIP32 *"v2"*, reemplazando al antiguo esquema **Daedalus (HD random)** de Byron que tenía problemas [15]. Todos los monederos oficiales actuales (Daedalus, Yoroi, etc.) convergen en CIP-1852 (Icarus sequential) para garantizar interoperabilidad [16]. *Nota:* Durante Byron, Yoroi (Icarus) ya utilizaba derivación secuencial 44'-Ed25519, mientras que Daedalus original usaba un método legacy (random indexes); hoy ambos han migrado al nuevo estándar [3] [17].

Construcción de direcciones Cardano: una vez derivadas las llaves, la dirección **no es simplemente la llave pública**. Cardano (era Shelley) utiliza un formato de **dirección Bech32** con estructura binaria interna definida en **CIP-19**. Una dirección **base (Shelley)** combina dos componentes: - Un **hash de la llave pública de pago** (28 bytes obtenidos mediante **Blake2b-224** sobre la pubkey) [18] [19]. - Un **hash de la llave pública de stake** (otros 28 bytes, Blake2b-224 de la pubkey de staking). - Un **byte de cabecera** que indica el tipo de dirección (base vs enterprise, etc.) y la red (mainnet = 1, testnet = 0) [20] [21].

Estas partes se codifican en Bech32 con prefijo `"addr"` (mainnet) o `"addr_test"` (testnet) según CIP-005 [22]. Por ejemplo, una **dirección base** mainnet inicia con "`addr1...`", mientras que una **dirección enterprise** (solo pago, sin stake) inicia con "`addr1v...`" – esta es más corta porque carece de la parte de stake [23] [20]. Internamente, para una dirección base, el header byte podría ser `0x01` (indicando pago y stake key presentes, mainnet) seguido del hash de pago y hash de stake [20]. Para una enterprise (solo pago) el header podría ser `0x61` (indica solo pago, mainnet) seguido del hash de pago [19]. Finalmente, la secuencia de bytes de la dirección se convierte a Bech32 (incluyendo sumas de comprobación) para obtener la cadena legible por humanos [22] [19].

En resumen, **para implementar en Rust** la derivación de Cardano desde la frase semilla, se necesitan los siguientes pasos y consideraciones:

- **BIP39**: usar una biblioteca o función para obtener la **seed** de 512 bits a partir de la mnemonic (por ejemplo, usando `bip39` crate).
- **Master key (XPrv)**: aplicar HMAC-SHA512 con key `"ed25519 cardano seed"` a la seed (según SLIP-0010/0023) [5]. Luego, ajustar los bits de la mitad izquierda (kL) como exige Ed25519 (clear los 3 bits inferiores, clear bit más alto y set segundo más alto) [7]. Conservar kL y kR como clave privada extendida (64 bytes) y la **chain code** (32 bytes). En Rust, se puede usar crates como `ed25519_bip32` que ya implementan este procedimiento de **Ed25519-BIP32**

compatible con Cardano [24]. De hecho, la referencia oficial de Cardano apunta a una implementación en Rust de BIP32-Ed25519 [24].

- **Derivación de hijos**: iterativamente derivar según la ruta m/1852'/1815'/... usando la función de derivación HD. Para hardened indices (>= 2^31), el algoritmo Ed25519-BIP32 utiliza el esquema privado->child (incluye 0x00 + priv + index en HMAC); para non-hardened, utiliza pubkey en la HMAC (es decir, soporta *soft derivation* como BIP32) [25]. Cabe mencionar que la mayoría de derivaciones en Cardano son hardened *hasta account*, luego soft para role/index, tal como se explicó. Si se usa una librería (p.ej. `ed25519_bip32` o la incluida en la crate *cardano*), esta manejará internamente el derivar hardened o no según el bit 31 del índice. Asegurarse de utilizar el **Derivation Scheme v2 (Icarus)** y **purpose=1852'** para Shelley – por ejemplo, muchas librerías Cardano (como *cardano-crypto*, *cardano-wallet*) tienen opciones para esquema Byron vs Shelley. En CIP-3 se documentan test vectors y distintos algoritmos (Byron Icarus, Ledger/Trezor, etc.), pero para nuevos desarrollos CIP-1852 es el estándar [17].
- **Obtener llaves públicas**: derivar la **clave pública extendida (XPub)** de cada XPrv hijo (normalmente librerías devuelven XPub directamente). Un XPub Cardano suele contener la clave pública (Ed25519 point, 32 bytes) y un chain code de 32 bytes [26] [27]. La obtención del XPub implica escalar la base Ed25519 con la clave privada escalar kL (lo cual las libs Ed25519 pueden hacer, e.j. usando `ed25519-dalek`).
- **Dirección**: para construir la dirección Shelley:
- Calcular el **hash** de 28 bytes (Blake2b-224) de la *clave pública de pago*. Igual para la *clave pública de staking* [18].
- Construir el **payload** de dirección: header + hashPago + [hashStake]. Ejemplo: para mainnet base address, header 0x01 [28]; para enterprise, header 0x61 [19]. (El esquema completo incluye otros tipos como Pointer o Enterprise script, detallados en CIP-19).
- **Codificar en Bech32** con el prefijo adecuado (`addr` para mainnet, `addr_test` para testnet). Rust tiene crates como `bech32` para esto, o se puede implementar usando la especificación BIP-173 [29] [30].

Con lo anterior, se podrá obtener la dirección Cardano final. Verificación: una dirección **base** generada debe empezar con *addr1...* y puede comprobarse en exploradores (CardanoScan, etc.) que corresponde a la payment key y stake key esperadas [21].

> **Nota:** Las implementaciones existentes pueden servir de referencia. Por ejemplo, *Cardano Serialization Library* (disponible en Rust y WASM) implementa estas derivaciones y construcciones de direcciones. También la crate `cardanoSharp` en C# sigue CIP-1852 (Shelley) y CIP-1853/1855 para otros tipos de keys [31]. En Rust, librerías como `ed25519-bip32` o `bip32` (con soporte SLIP-10) facilitan la derivación Ed25519; y para hashing Blake2b-224 se puede usar crates como `blake2b_simd`. Siempre asegúrate de que la librería aplica la corrección de bits adecuada (Ed25519 clamping y bit extra de compatibilidad Cardano) – por ejemplo, SLIP-0023 requiere limpiar un bit adicional ("third highest bit") de la clave para compatibilidad con derivación de Cardano [7] (esto está incorporado en las implementaciones usadas por Yoroi/Daedalus).

Solana – Derivación HD desde frase semilla (BIP44 / SLIP-0010)

En **Solana**, la derivación de cuentas también se basa en BIP39/BIP44 con Ed25519, pero con algunas convenciones propias de la comunidad. Solana adopta el coin type `501` (asignado en SLIP-44) y **usa solamente derivaciones *hardened*** en todos los niveles de su ruta estándar [32] [33]. Esto significa que, a diferencia de Cardano, **no hay derivación pública (soft)** en la práctica común de Solana – todas las derivaciones incluyen el sufijo `'`.

Mnemonic a master key: Se utiliza BIP39 para obtener la seed de 512 bits desde la frase. Luego, se aplica **SLIP-0010** (Ed25519) para derivar la clave maestra. En SLIP-0010, la seed se procesa con HMAC-SHA512 usando la clave `"ed25519 seed"` [5] (a diferencia de Cardano, Solana no usa una variante especial como `"cardano seed"`; se adhiere al esquema general de ed25519). El resultado de 64 bytes se divide en: 32 bytes de clave privada maestra (`kL`, tras aplicar clamping Ed25519 estándar) y 32 bytes de chain code. Dado que Solana *no* realiza derivaciones no-hardened, realmente la chain code es utilizada solo si se derivan subllaves adicionales, pero en la práctica muchos softwares simplemente toman esos 32 bytes de chain code y los mantienen para derivar hijos hardened. De hecho, algunas implementaciones simples (como ciertos ejemplos con `solana-web3.js`) derivan directamente la primera clave privada de 32 bytes de la seed sin usar chain code cuando solamente necesitan la cuenta 0 [34] [35], pero para soportar múltiples cuentas es necesario hacer la derivación HD completa con chain code.

Derivation path estándar: La comunidad de Solana convergió principalmente en dos rutas BIP44 posibles: - `m/44'/501'/0'/0'` – es la ruta **más utilizada** por los monederos *browser* (como **Phantom**, **Sollet**, etc.) para la primera cuenta [36] [37]. - `m/44'/501'/0'` – era una variante usada por el monedero **Solflare** (en modos legacy) para la cuenta principal [37].

Adicionalmente, la herramienta oficial de línea de comando **Solana CLI** por defecto genera la clave usando solo el prefijo `m/44'/501'` (es decir, toma la *root key* en coin type 501 directamente como cuenta) [36] [37]. Esto ocasiona que una misma mnemonic importada en la CLI dé una dirección diferente que en Phantom, a menos que se especifique la misma derivation path. En general, **Phantom/Sollet** (ruta de 4 niveles hardened) se consideran el *estándar de facto* para usuarios, por lo que es recomendable implementar esa por defecto.

Desglose de `m/44'/501'/0'/0'`: - `44'` es el **purpose BIP44** estándar [38]. - `501'` es el **coin type** Solana (501) [38]. - `0'` (tercer nivel) típicamente representa el **número de cuenta**. Los monederos permiten cambiar este índice para generar nuevas **direcciones adicionales** bajo la misma seed (Phantom, por ejemplo, incrementa este valor para crear múltiples cuentas en la cartera) [39]. - El último `0'` a veces es referido como **change** o **derivación adicional**, pero en Solana *siempre se usa 0'* ya que no se distingue entre direcciones externas/internas (Solana es modelo **account-based**, no UTXO) [40]. En la práctica, este cuarto nivel no varía y permanece en 0' para todas las direcciones "normales". *(Algunos desarrolladores omiten este nivel y solo usan 3 niveles hardened, de ahí la ruta de Solflare* `m/44'/501'/0'` *que conceptualmente fija el change=0 implícitamente.)*

Importante: **todas las posiciones llevan** `'`. Aunque en la notación BIP44 tradicional los últimos niveles (change, address_index) no serían hardened, en Solana igualmente se tratan como hardened. De hecho, muchas librerías/SDK de Solana *ignoran* si escribes o no el `'` en esos niveles, automáticamente tratan cualquier índice como hardened (puesto que Ed25519 no soporta derivación pública fácilmente) [32] [41].

Derivación de hijos hardened: La derivación hardened en Ed25519 (SLIP-0010) funciona pasando **0x00 || parent_privkey || index** al HMAC-SHA512 para obtener el hijo [25]. Por ello se requiere la clave privada del padre para derivar cada nivel (no es posible derivar hijos solo con la pubkey). En Rust, podemos usar librerías de HD wallet genéricas con soporte Ed25519: - Por ejemplo, crate `bip32` con feature de Ed25519 (usa *slip10*) o la mencionada [`ed25519_bip32`]. - Existe también `slip10` crate para derivación según SLIP-0010. - La crate `solana-sdk` de Solana posee utilidades: p.ej. estructura `DerivationPath` [42] para parsear rutas, aunque la derivación en sí suele delegarse a funciones de su dependencia *ed25519-dalek*.

Tras derivar `m/44'/501'/0'/0'`, obtendremos la **clave privada** de la cuenta. De ella, la **clave pública** se obtiene multiplicando por el generador de Ed25519 (las librerías como *ed25519-dalek* pueden generar la pubkey a partir de un secret key de 32 bytes fácilmente).

Formato de dirección Solana: En Solana, la **dirección** de una cuenta es simplemente su **clave pública de 32 bytes** codificada en **Base58** [43] [44] . No hay prefijos humanos ni variantes testnet/mainnet en la codificación (Solana distingue redes a nivel de RPC, no por el formato de dirección). El alfabeto Base58 usado excluye caracteres visualmente confusos (`0` , `I` , `O` , `l`) [43] . Prácticamente cualquier cadena de 44 caracteres Base58 puede ser una dirección Solana válida, pero obviamente solo las correspondientes a claves Ed25519 conocidas tendrán claves privadas asociadas. No se añade un checksum explícito a la dirección; la verificación de integridad se da por la longitud fija y el propio uso de Base58Check implícito en muchas libs. Por ejemplo, una clave pública en bytes podrá representarse como algo como `5A4...XYZ` en base58.

Uso en Rust: Para implementar, se resumen los pasos: - Obtener seed de 64 bytes con BIP39 (por ejemplo con `bip39::Mnemonic`). - Usar HMAC-SHA512 con `"ed25519 seed"` (puede hacerse manualmente con crate `hmac` + `sha2` or `sha2::Sha512` , o usando crate `slip10` que ya realiza `Slip10Key::derive_from_path(seed, Ed25519, path)`). - Derivar secuencialmente los índices hardened de la ruta deseada: - Derivar `44'` del master, luego `501'` , luego cuenta `0'` , luego cambio `0'` . Cada derivación produce un nuevo key + chain code. - Tomar la clave privada resultante (32 bytes) y generar la clave pública (32 bytes). En Rust, `ed25519-dalek` crate puede crear a partir de `[u8;32]` un `Keypair` o `PublicKey` . - Codificar la pubkey en Base58. Para esto se puede usar la crate `bs58` fácilmente.

Variantes de derivación en Solana: Si se quiere ser exhaustivo, se puede soportar las variantes: - **Phantom/Sollet (m/44'/501'/n'/0')**: la principal – usar `n'` como índice de cuenta para múltiples cuentas. Phantom permite crear “Cuenta 1”, “Cuenta 2”, etc., que corresponden a `0'` , `1'` , etc. en ese tercer nivel [39] . - **Solflare (m/44'/501'/n')**: esencialmente igual que la anterior pero omitiendo el nivel final (si implementamos la de Phantom, la de Solflare es simplemente usar siempre `change=0'` y luego *no derivar* ese nivel, o conceptualmente es lo mismo si se piensa que Solflare consideraba `m/44'/501'/0'/0'` pero con el último `'0'` implícito). - **CLI (m/44'/501')**: es solo el key raíz de coin type. No es recomendable usarlo en apps, pero es bueno saberlo. (En caso de necesitar reproducir direcciones CLI, sería la clave derivada después de `501'` – equivaldría a account=0' & change=0' por omisión en esa herramienta).

Sin embargo, la **mayoría de usuarios** de Solana esperan la ruta Phantom, por lo que es la que conviene utilizar por defecto [37] . De hecho, muchos monederos (Exodus, Ledger Live después de actualizaciones) adoptaron `m/44'/501'/0'/0/0` *con el último índice sin hardening* [45] , aunque internamente debido a la implementación slip-10, ese `0/0` probablemente se interpreta como `0'/0'` . Por simplicidad, enfocarse en la convención Phantom garantiza compatibilidad amplia.

Ejemplo: con la frase *"neither lonely flavor argue grass remind eye tag avocado spot unusual intact"* (12 palabras), la derivación `m/44'/501'/0'/0'` debería dar una clave pública que comienza con, digamos, `5n9...` y es la dirección que Phantom/Sollet mostrarían [46] [47] . Derivar la misma frase con la ruta CLI `m/44'/501'` resultaría en otra dirección totalmente distinta [36] , lo que ilustra la importancia de usar la ruta correcta.

Recursos de Rust: Existe una crate utilitaria llamada `solana_derivation_path` [42] que define rutas BIP44 de Solana y puede integrarse con herramientas Solana. Adicionalmente, *Solana SDK* (en Rust) incluye funcionalidad para convertir mnemonics BIP39 a keypairs. Por ejemplo, la función

`solana_cli_config::Keypair::from_seed_phrase` (en el código fuente de Solana CLI) usa la librería *tiny-bip39* y *ed25519-dalek* para derivar según la ruta especificada. Revisar esas implementaciones puede ser instructivo. En cualquier caso, implementar manualmente con las crates estándar HMAC + Sha512 + ed25519 no es excesivamente complejo dado que solo se trata de aplicar la fórmula de derivación hardened repetidamente.

En resumen, **Solana** utiliza un esquema HD **BIP44/SLIP-0010** Ed25519 simplificado (todos índices hardened) y su dirección final es simplemente la **clave pública Ed25519 en base58** [18] [44] . Mantener consistencia con la ruta esperada (Phantom) es crucial para que las direcciones generadas coincidan con las de los monederos populares [36] .

Conclusiones

Tanto Cardano como Solana requieren entender las particularidades de **Ed25519 HD derivation** para implementar correctamente la derivación de direcciones en Rust. En Cardano, seguir **CIP-1852** con la variante **Ed25519-BIP32** (y sus ajustes de bits y paths específicos) garantiza compatibilidad con Yoroi/Daedalus y la generación de direcciones `addr1...` correctas [8] [22] . En Solana, aplicar **SLIP-0010** con la ruta `m/44'/501'/0'/0'` produce las direcciones base58 esperadas por la mayoría de wallets (Phantom, etc.) [36] [37] .

Aprovechar librerías existentes en Rust (p. ej. `ed25519-bip32`, `bip32`, `slip10`, `blake2`, `bs58`) puede simplificar la implementación, pero es fundamental consultar la documentación oficial y CIPs para asegurarse de los detalles (como la diferencia entre `'44'` vs `'1852'`, o el manejo de bits de Ed25519). Los enlaces citados proporcionan documentación de referencia y ejemplos que respaldan estos procesos: - CIP de Cardano (CIP-1852, CIP-003, CIP-019) para especificaciones de HD wallets y formato de direcciones [2] [22] . - Discusiones en foros y documentación de IOHK (*Adrestia*) que explican diferencias Byron/Shelley e ilustran la jerarquía de derivación [48] [13] . - Artículos y recursos de Solana que aclaran las derivation paths soportadas por diferentes wallets y cómo derivar múltiples cuentas [36] [37] .

Con esta investigación, se cuenta con bases sólidas para implementar o corregir la derivación de direcciones Cardano y Solana en Rust, asegurando que a partir de la frase semilla se obtengan exactamente las mismas direcciones que un usuario esperaría en los monederos oficiales.

Referencias utilizadas:

- Cardano Improvement Proposal CIP-1852 – *HD Wallets for Cardano* [49] [10] .
- Cardano Improvement Proposal CIP-3 – *Wallet Key Generation* (BIP39, ED25519-BIP32, historial Byron vs Shelley) [4] [6] .
- Discusión técnica: *Public Key to Shelley Address* – Cardano Forum (ejemplo de construcción de dirección Bech32 a partir de llaves) [22] [19] .
- Documentación Adrestia (IOHK) – *Address Derivation*, diferencias BIP44 vs Cardano, esquema HD Sequential [48] [13] .
- Artículo de Nick Frostbutter – *Derive Solana Addresses* (explicación de mnemonic -> derivation path -> address en Solana) [36] [50] .
- Publicación en Medium – *Derivation paths on Solana* (resumen de rutas Phantom/Solflare/CLI) [37] [51] .
- Solana Cookbook – *Restore Keypair from Mnemonic* (ejemplos de código JS/TS para derivación BIP44 ed25519) [46] [47] .

- Crate Rust `ed25519_bip32` – documentación (explica adaptación de BIP32 a Ed25519 y tamaños de claves extendidas) [9] [26].
- Issue SLIP-0023 (Cardano seed) – notas sobre ajustes de bits adicionales para compatibilidad Cardano [7].
- SLIP-0010 – Ed25519 derivation standard (usado en Solana for master key, referenciado en CIP-3) [52] [5].

---

[1] [2] [3] [8] [10] [11] [15] [16] [49] CIP-1852 | HD (Hierarchy for Deterministic) Wallets for Cardano
https://cips.cardano.org/cip/CIP-1852

[4] [6] [17] [24] [52] CIP-3 | Wallet Key Generation
https://cips.cardano.org/cip/CIP-3

[5] slips/slip-0023.md at master · satoshilabs/slips - GitHub
https://github.com/satoshilabs/slips/blob/master/slip-0023.md

[7] SLIP-0023 : Modification Proposal · Issue #827 · satoshilabs/slips
https://github.com/satoshilabs/slips/issues/827

[9] [14] [25] [26] [27] ed25519_bip32 - Rust
https://docs.rs/ed25519-bip32/latest/ed25519_bip32/

[12] [13] [48] Address Derivation – Adrestia Project Docs
https://input-output-hk.github.io/adrestia/cardano-wallet/concepts/address-derivation

[18] [19] [20] [21] [22] [23] [28] [29] [30] Public Key to Shelly Address - Misc Dev Talk - Cardano Forum
https://forum.cardano.org/t/public-key-to-shelly-address/114553

[31] Derive and Create Keys - CardanoSharp
https://cardanosharp.com/docs/keys

[32] [33] [36] [38] [39] [40] [41] [43] [44] [50] Derive Solana Addresses – Nick Frostbutter
https://nick.af/articles/derive-solana-addresses

[34] [35] [46] [47] How to Restore a Keypair from a Mnemonic | Solana
https://solana.com/developers/cookbook/wallets/restore-from-mnemonic

[37] [51] Derivation paths on Solana. Article available on our blogging site... | by JW | Medium
https://medium.com/@josh.wolff.7/derivation-paths-on-solana-fb08d3dd09f1

[42] solana_derivation_path - Rust - Docs.rs
https://docs.rs/solana-derivation-path

[45] Derivation paths in Exodus
https://support.exodus.com/support/en/articles/8598933-derivation-paths-in-exodus