Generating and Handling Extractable Ed25519 CryptoKeyPairs in the Solana v2.x Ecosystem

I. Introduction

The Solana blockchain platform relies fundamentally on public-key cryptography, specifically the Ed25519 signature scheme, for account identification and transaction authorization.[1] Traditionally, Solana development libraries like @solana/web3.js (version 1.x) provided straightforward mechanisms for keypair generation where the secret key material was directly accessible as a byte array.[3] However, with the evolution towards Solana SDK v2.x and libraries such as @solana/keys, there is an increasing alignment with web standards, notably the Web Crypto API. This API introduces the CryptoKeyPair and CryptoKey objects, which encapsulate cryptographic keys and offer more granular control over their properties, including "extractability".[4]

This report details various approaches for generating Ed25519 CryptoKeyPairs where the private key is "extractable" within the Solana v2.x JavaScript/TypeScript ecosystem, focusing on @solana/keys and related tooling. An extractable key is one whose raw material can be exported from the CryptoKey object, a capability crucial for use cases such as wallet backup, key migration, or interoperability with systems requiring raw key bytes.[5] The ability to extract key material, while offering flexibility, also introduces significant security responsibilities that must be carefully managed. This analysis will explore the methods, their implications, and best practices for handling such keys.

II. Understanding Key Concepts

A clear understanding of foundational cryptographic concepts and the specific APIs involved is essential before delving into generation methods.

A. Solana Keypairs and their Role

In Solana, a keypair consists of a public key and a corresponding secret (private) key.[1]

- The **public key** serves as the unique address of an account on the Solana network. It can be freely shared and is used to identify recipients for transactions or to locate on-chain accounts.[1]
- The **secret key** (or private key) is used to prove ownership and authorize actions related to the account, such as signing transactions to transfer tokens or interact with smart contracts. As its name implies, the secret key must be kept confidential; anyone possessing the secret key has control over the associated

account and its assets.[1]

The security of the entire Solana account model hinges on the secrecy of the private key.

B. CryptoKeyPair in Web Crypto API

The Web Crypto API, a standard interface for performing cryptographic operations in web applications, defines the CryptoKeyPair object.[5] A CryptoKeyPair is a dictionary-like object typically containing two properties:

- publicKey: A CryptoKey object representing the public key.
- privateKey: A CryptoKey object representing the private key.

The CryptoKey objects themselves store the key material along with metadata about the key, such as the algorithm it's intended for and its permitted usages (e.g., signing, encryption).[5] Libraries within the Solana v2.x ecosystem, such as @solana/keys, now utilize these CryptoKey objects, promoting standardization and leveraging browser-native or Node.js-native cryptographic implementations.[4]

C. The "Extractable" Flag

A critical property of a CryptoKey object, particularly relevant to this report, is its extractability. When generating a key or key pair using crypto.subtle.generateKey(), one of the parameters is a boolean flag named extractable.[5]

- If extractable is set to true, the key material can be exported from the CryptoKey object using methods like crypto.subtle.exportKey() or crypto.subtle.wrapKey().
- If extractable is set to false (which is often the default for security reasons in many implementations if not specified), the key material is designed to be non-exportable, meaning it cannot be directly retrieved by the application code.[5]

The ability to generate a CryptoKeyPair with an extractable private key is the central focus of the user's query.

D. @solana/keys (web3.js v2.x context)

The @solana/keys package is a utility library designed for validating, generating, and manipulating addresses and cryptographic key material within the Solana ecosystem.[4] It is a component of the broader @solana/kit, which bundles various libraries for Solana development. A key feature of @solana/keys in the v2.x context is its adoption of the Web Crypto API's CryptoKey objects for representing key material, moving away from the raw byte arrays used more directly in older versions of @solana/web3.js.[4]

This aligns Solana development more closely with web standards for cryptography.

III. Methods for Generating Extractable CryptoKeyPairs

Several methods can be employed to generate Ed25519 CryptoKeyPairs with extractable private keys, ranging from direct Web Crypto API usage to higher-level abstractions provided by Solana libraries.

A. Direct Use of Web Crypto API: crypto.subtle.generateKey()

The most fundamental way to generate an extractable CryptoKeyPair is by directly using the crypto.subtle.generateKey() method provided by the Web Crypto API. This method offers granular control over key generation parameters. For Ed25519, the parameters are [5]:

1. algorithm: An object specifying the algorithm. For Ed25519, this can be the string "Ed25519" or an object { name: "Ed25519" }.
2. extractable: A boolean. To make the private key extractable, this must be set to true.
3. keyUsages: An array of strings defining what the key can be used for. For Ed25519, typical usages are ["sign", "verify"].

Example Code:

TypeScript

```typescript
async function generateExtractableEd25519KeyPairWebCrypto(): Promise<CryptoKeyPair> {
  const keyPair = await crypto.subtle.generateKey(
    { name: "Ed25519" },
    true, // Crucial: makes the private key extractable
    ["sign", "verify"]
  );
  return keyPair;
}

// Usage:
// generateExtractableEd25519KeyPairWebCrypto().then(keyPair => {
//   console.log("Public key type:", keyPair.publicKey.type);
//   console.log("Private key type:", keyPair.privateKey.type);
//   console.log("Private key extractable:", keyPair.privateKey.extractable);
// });
```

This approach provides maximum control and adheres closely to web standards. It's important to note that native support for Ed25519 in crypto.subtle might not be available in all JavaScript environments (especially older Node.js versions or some browsers). In such cases, a polyfill like @solana/webcrypto-ed25519-polyfill can be used to enable Ed25519 operations by proxying calls to a userspace implementation.[6] However, this polyfill explicitly warns that it cannot guarantee non-exportability for keys generated in userspace, even if extractable is set to false.[6] For generating *extractable* keys, this limitation is less of a concern, but awareness of the underlying mechanism (native vs. polyfill) is beneficial.

B. Using @solana/keys Functions

The @solana/keys library provides functions that wrap or utilize Web Crypto API operations.

1. generateKeyPair() from @solana/keys
   The generateKeyPair() function in @solana/keys is designed to generate an Ed25519 public/private key pair, returning a Promise<CryptoKeyPair>.4
   TypeScript
   ```
   import { generateKeyPair } from '@solana/keys';
   // const { privateKey, publicKey } = await generateKeyPair();
   ```

   However, this function, as per its documented signature, does *not* accept an extractable option as a direct parameter.[4] The extractability of the CryptoKey objects it produces would therefore depend on the default behavior of the underlying crypto.subtle.generateKey call it makes, or if @solana/keys internally sets a specific extractability flag. If the underlying default is false, or if the library enforces non-extractability for security reasons, then this function would not reliably produce an *extractable* private key. For guaranteed extractability when generating a *new* keypair, direct use of crypto.subtle.generateKey with extractable: true is more explicit.
2. createPrivateKeyFromBytes(bytes, extractable: true) followed by getPublicKeyFromPrivateKey(privateKey, extractable?: true)
   This two-step approach using @solana/keys is suitable when one already possesses raw private key bytes (e.g., from a seed, a previously generated key, or a legacy system) and needs to construct an extractable CryptoKey (and subsequently a CryptoKeyPair).
   - createPrivateKeyFromBytes(secretKeyBytes: Uint8Array, extractable?: boolean): This function takes a 32-byte Uint8Array representing the private

key and an optional boolean extractable flag. Setting extractable to true ensures the resulting CryptoKey object for the private key is extractable.[4]

- getPublicKeyFromPrivateKey(privateKey: CryptoKey, extractable?: boolean): Given an extractable private CryptoKey, this function derives the corresponding public CryptoKey. The optional extractable flag here pertains to the resulting public key.[4]

Example Code: TypeScript

```typescript
import { createPrivateKeyFromBytes, getPublicKeyFromPrivateKey } from '@solana/keys';
import { getBytesEncoder } from '@solana/codecs-core'; // For example, if using codecs

async function createExtractableKeyPairFromExistingBytes(privateKeyBytes: Uint8Array):
Promise<CryptoKeyPair> {
  if (privateKeyBytes.length!== 32) {
    throw new Error("Private key bytes must be 32 bytes long for Ed25519.");
  }

  const privateCryptoKey = await createPrivateKeyFromBytes(privateKeyBytes, true); //
Ensure private key is extractable
  const publicCryptoKey = await getPublicKeyFromPrivateKey(privateCryptoKey, true); //
Public key extractability

  return { privateKey: privateCryptoKey, publicKey: publicCryptoKey };
}

// Example usage with placeholder bytes:
// const examplePrivateKeyBytes = new Uint8Array(32).fill(1); // Replace with actual 32-byte private key
// createExtractableKeyPairFromExistingBytes(examplePrivateKeyBytes).then(keyPair => {
//     console.log("Created extractable CryptoKeyPair from bytes.");
//     console.log("Private key extractable:", keyPair.privateKey.extractable);
// });
```

This method is the designated path within @solana/keys for creating an explicitly extractable CryptoKey when starting from raw byte material.

C. Abstractions in Higher-Level Libraries (e.g., gill, @solana/kit)

Higher-level libraries in the Solana ecosystem often provide abstractions over these fundamental key generation mechanisms.

1. @solana/kit and generateKeyPairSigner()
   @solana/kit is a comprehensive toolkit that bundles several lower-level packages, including @solana/keys and @solana/signers.4 The @solana/signers package introduces the KeyPairSigner interface, an abstraction for signing messages and transactions using a CryptoKeyPair.7 The function generateKeyPairSigner() is provided to create such a signer.
   The Solana Cookbook shows an example for creating an "Extractable keypair"

using generateKeyPairSigner from @solana/kit.[8] However, the documentation for generateKeyPairSigner within the context of the gill library (a successor/evolution of parts of @solana/kit) states that its default generateKeyPairSigner() creates *non-extractable* signers for enhanced security.[9] The cookbook snippet for @solana/kit does not show an explicit extractable flag being passed to generateKeyPairSigner [8], and it's not definitively stated whether this function internally ensures extractability by default.[8] This suggests potential variability or evolution in behavior.

2. gill library's generateExtractableKeyPairSigner()
   To address the need for explicitly extractable keypairs at a higher level of abstraction, the gill library (developed by Anza, formerly Solana Labs) provides a distinct function: generateExtractableKeyPairSigner().9 This function is explicitly designed to return a KeyPairSigner whose underlying CryptoKeyPair has an extractable private key.
   Example Code (Conceptual, based on gill documentation):
   TypeScript

```
import { generateExtractableKeyPairSigner } from "gill"; // Assuming 'gill' is used

// async function generateGillExtractableSigner() {
//     const signer = await generateExtractableKeyPairSigner();
//     // The 'signer' object is a KeyPairSigner.
//     // Its underlying CryptoKeyPair's private key is extractable.
//     // To get the CryptoKey objects, one might need to inspect the signer's properties
//     // or use methods it provides, if any, for accessing the raw CryptoKeyPair.
//     // The exact structure of KeyPairSigner and how it exposes the CryptoKeyPair
//     // would be defined by the 'gill' or '@solana/signers' library. [9]
//     console.log("Generated extractable KeyPairSigner with gill.");
// }
```

   The gill documentation strongly warns that using extractable keypairs is inherently less secure and should only be done when there is an explicit need to extract the key material, such as saving it to a file.[9] The KeyPairSigner itself is an abstraction; to access and export the underlying CryptoKey material, one would still need to use crypto.subtle.exportKey() on the privateKey object of the CryptoKeyPair that the KeyPairSigner presumably holds. The precise mechanism for accessing this CryptoKeyPair from the KeyPairSigner instance would depend on the library's API design.[9]

IV. Legacy @solana/web3.js Keypair.generate() (v1.x)

For context, it is useful to briefly contrast the modern CryptoKeyPair approach with the key generation mechanism in the legacy @solana/web3.js v1.x library. The

Keypair.generate() static method in this older library returns an instance of a Keypair class.[1] This Keypair object directly exposes two properties:

- publicKey: An instance of PublicKey.
- secretKey: A Uint8Array of 64 bytes, representing the concatenation of the 32-byte private key and the 32-byte public key.[3]

Example Code (v1.x):

TypeScript

```
// import { Keypair } from "@solana/web3.js"; // v1.x

// const keypair_v1 = Keypair.generate();
// console.log(`The public key is: `, keypair_v1.publicKey.toBase58());
// console.log(`The secret key (Uint8Array) is: `, keypair_v1.secretKey);
// // The raw secret key bytes are directly accessible via keypair_v1.secretKey
```

In this model, the "extractability" of the private key is inherent because the secretKey property provides direct access to the raw byte array containing the private key material.[1] There is no separate extractable flag or exportKey mechanism as seen with the Web Crypto API's CryptoKey objects. This direct access simplifies obtaining the key bytes but offers less of the structured security controls (like non-extractable keys by default) that the Web Crypto API aims to provide. The shift to CryptoKeyPair in v2.x tooling represents an adoption of these more robust, standardized cryptographic primitives.

V. Exporting and Handling Extractable CryptoKey Material

Once an extractable CryptoKeyPair is generated, the next step is often to export the key material for storage, backup, or use in other systems.

A. Using crypto.subtle.exportKey()

The standard Web Crypto API method for exporting key material from an extractable CryptoKey is crypto.subtle.exportKey(format, key).[5]

- format: A string specifying the desired export format. Common formats include:
  - 'raw': Exports the raw key bytes. For Ed25519, this typically yields a 32-byte ArrayBuffer for a public key. Exporting Ed25519 private keys in 'raw' format

directly is often not supported; they are usually exported as 'pkcs8' first.[11]
    - 'pkcs8': A standard ASN.1 format for private keys. The output is an ArrayBuffer containing the DER-encoded PKCS#8 structure.
    - 'spki': SubjectPublicKeyInfo, a standard ASN.1 format for public keys. The output is an ArrayBuffer.
    - 'jwk': JSON Web Key, a JSON representation of the key.
- key: The CryptoKey object to export.

Example Code for Exporting Raw Bytes:

TypeScript

```typescript
async function exportPrivateKeyRawBytes(privateKey: CryptoKey): Promise<Uint8Array> {
  if (!privateKey.extractable) {
    throw new Error("Private key is not extractable.");
  }
  // Ed25519 private keys are typically exported as PKCS#8, then parsed.
  // The raw 32-byte private key is usually the last 32 bytes of the PKCS#8 structure for Ed25519.
  const exportedPkcs8 = await crypto.subtle.exportKey("pkcs8", privateKey);
  // The slice operation assumes a specific structure of Ed25519 PKCS#8.
  // For Ed25519, the key itself is an OCTET STRING, often at the end of the structure.
  // A more robust method might involve an ASN.1 parser, but.slice(-32) is a common heuristic.[11]
  const rawPrivateKeyBytes = new Uint8Array(exportedPkcs8.slice(-32));
  return rawPrivateKeyBytes;
}

async function exportPublicKeyRawBytes(publicKey: CryptoKey): Promise<Uint8Array> {
  // Public keys are often considered extractable by default for 'raw' export,
  // or their 'extractable' flag might not restrict 'raw' export.
  // However, if it was explicitly created as non-extractable, this could fail.
  // For consistency, one might ensure it was created with extractable:true if issues arise.
  const exportedRaw = await crypto.subtle.exportKey("raw", publicKey);
  return new Uint8Array(exportedRaw);
}
```

The choice of format depends on the requirements of the system that will consume the exported key. For many Solana-related purposes, such as importing into wallets or using with command-line tools, raw byte arrays are needed.

B. Converting Key Material to Common Solana Formats (e.g., Base58)

Solana ecosystem tools and wallets often expect secret keys in specific byte formats, commonly encoded in Base58. For instance, Phantom wallet and the Solana CLI typically use a 64-byte secret key format, which is the concatenation of the 32-byte private key followed by the corresponding 32-byte public key, all then Base58 encoded.[10]

Example Code (Conceptual, for Phantom-compatible format):

TypeScript

```
import { encode as bs58Encode } from 'bs58'; // Common Base58 encoding library
// Alternatively, using @solana/codecs-strings:
// import { getBase58Encoder } from '@solana/codecs-strings';
// const base58Encoder = getBase58Encoder();

async function formatSecretKeyForSolanaWallet(privateCryptoKey: CryptoKey, publicCryptoKey:
CryptoKey): Promise<string> {
  const privateKeyBytes = await exportPrivateKeyRawBytes(privateCryptoKey); // 32
bytes
  const publicKeyBytes = await exportPublicKeyRawBytes(publicCryptoKey);   // 32 bytes

  if (privateKeyBytes.length!== 32 |
| publicKeyBytes.length!== 32) {
    throw new Error("Invalid key lengths after export.");
  }

  const combinedBytes = new Uint8Array(64);
  combinedBytes.set(privateKeyBytes, 0);
  combinedBytes.set(publicKeyBytes, 32);

  const base58EncodedSecretKey = bs58Encode(combinedBytes);
  // Using @solana/codecs-strings:
  // const base58EncodedSecretKey = base58Encoder.encode(combinedBytes);
  return base58EncodedSecretKey;
}
```

This conversion process bridges the gap between the structured CryptoKey objects from the Web Crypto API and the byte-oriented, Base58-encoded formats prevalent in the Solana ecosystem. Successfully performing this transformation is key to ensuring interoperability. The need to manually extract, combine, and re-encode key material highlights that simply having an "extractable CryptoKey" is an intermediate step; further processing is often required to make it usable with existing Solana tools and wallets.

C. Security Best Practices for Managing Exported Private Keys

The ability to extract private key material significantly increases security risks if not handled with extreme care. Once exported, the security of the key is entirely dependent on how it is stored and managed.

- **Extreme Caution:** Private key material grants complete control over the associated account and its assets.[1] Treat it as highly sensitive information.
- **Minimize Exposure:** Handle raw private key bytes in memory for the shortest duration necessary. Clear them from memory as soon as they are no longer needed, if possible (though JavaScript's garbage collection makes explicit clearing complex).
- **Secure Storage:** If exported keys must be stored, use strong encryption mechanisms (e.g., AES-256-GCM) with robust key derivation functions for the encryption key. Store encrypted keys in secure locations with strict access controls.
- **Avoid Hardcoding:** Never embed private keys directly in source code. Use secure methods for providing keys at runtime, such as environment variables (loaded via .env files for local development, but managed securely in production), or dedicated secrets management systems.[1]
- **User Responsibility:** If generating keys for end-users that they will export, clearly and strongly communicate that they are solely responsible for the security of their exported keys. Provide guidance on safe storage practices.
- **Non-Extractable by Default:** The gill library's approach of making keypair signers non-extractable by default and requiring an explicit function call for extractable ones is a good security posture.[9] This principle should be broadly applied: prefer non-extractable keys unless extractability is an explicit and unavoidable requirement.

The convenience afforded by extractable keys must be weighed against the substantial security burden they impose.

VI. Comparative Analysis and Recommendations

Choosing the right method for generating and handling extractable CryptoKeyPairs depends on the specific context and requirements.

A. Table: Comparison of Extractable CryptoKeyPair Generation Methods

Method/Function	Primary Library/API	Output Type	Control over Extractability	Ease of Use	Key Use Cases
crypto.subtle.generateKey()	Web Crypto API	Promise<CryptoKeyPair>	Direct via extractable: true flag	Lower-level, verbose, full control	New key generation, adherence to web standards
@solana/keys: createPrivateKeyFromBytes() + getPublicKeyFromPrivateKey()	@solana/keys	Promise<CryptoKey> (for private), then Promise<CryptoKey> (for public)	Explicit extractable: true parameter for private key	Mid-level, specific to existing key bytes	Creating CryptoKey objects from existing raw private key bytes, ensuring extractability
gill: generateExtractableKeyPairSigner()	gill (Anza)	Promise<KeyPairSigner> (wraps CryptoKeyPair)	Explicit function name indicates extractability	Higher-level, concise, part of gill ecosystem	Simplified generation of extractable signers within the gill framework
@solana/keys: generateKeyPair()	@solana/keys	Promise<CryptoKeyPair>	Indirect (depends on underlying crypto.subtle default)	Mid-level	General new keypair generation; extractability not guaranteed by function

					signature

B. Table: Common CryptoKey Export Options via exportKey() for Ed25519

exportKey() Format	**Applicable Key Type(s)**	**Typical Output for Ed25519**	**Common Use / Notes**
'pkcs8'	Private Key	ArrayBuffer (ASN.1 DER structure containing the private key and algorithm identifier)	Standard format for private key interchange. Raw 32-byte Ed25519 private key often parsed from this structure (e.g., last 32 bytes).[11]
'spki'	Public Key	ArrayBuffer (ASN.1 DER structure, SubjectPublicKeyInfo)	Standard format for public key interchange.
'raw'	Public Key	ArrayBuffer (Raw 32-byte public key)	Direct access to public key bytes. Widely used for addresses, identifiers.
'raw'	Private Key	Often *not directly supported* for Ed25519 private keys via exportKey. Obtain via pkcs8.	If supported, would be raw 32-byte private key. More common to get from parsing 'pkcs8' export.
'jwk'	Both	Object (JSON Web Key format, includes key type, curve, x/y coords for public, d for private)	Self-describing JSON format, useful for web applications and some protocols. Includes metadata along with key material.

C. Guidance on Choosing the Appropriate Method

- **For generating new, extractable Ed25519 CryptoKeyPairs with maximum control and adherence to web standards:** Directly use crypto.subtle.generateKey({ name: "Ed25519" }, true, ["sign", "verify"]). This provides the most explicit control over extractability.
- **When starting from existing raw 32-byte private key material (e.g., from a seed or legacy system) and needing an extractable CryptoKey representation:** Use @solana/keys' createPrivateKeyFromBytes(bytes, true) to create the private CryptoKey, followed by getPublicKeyFromPrivateKey() to derive the public CryptoKey.
- **If working within the gill library ecosystem and preferring a higher-level abstraction for creating an extractable signer:** Use gill's generateExtractableKeyPairSigner(). Be mindful of the security warnings associated with this.[9]
- **Regarding @solana/keys' generateKeyPair() or @solana/kit's generateKeyPairSigner():** Exercise caution if guaranteed extractability is required. Unless their documentation explicitly confirms they use extractable: true internally or the specific JavaScript environment's crypto.subtle.generateKey defaults to extractable for Ed25519, these may produce non-extractable keys by default. Always verify the extractable property of the resulting privateKey if relying on these for extractable keys.

D. Summary of Security Considerations

The generation and handling of extractable private keys carry inherent security risks.

- **Prefer Non-Extractable Keys:** If the use case does not strictly require the private key material to be exported, always opt for non-extractable keys. This is the more secure default, as it keeps the private key material confined within the cryptographic module (e.g., browser's SubtleCrypto implementation).
- **Responsibility:** The moment a private key is marked as extractable and its material is exported, the responsibility for its security shifts entirely to the application developer and, ultimately, the user.
- **Secure Handling is Paramount:** Implement robust measures for the protection of any exported private key material, including secure storage, encryption, and minimization of exposure.

VII. Conclusion

Generating extractable Ed25519 CryptoKeyPairs in the Solana v2.x ecosystem, particularly with libraries like @solana/keys, involves leveraging the Web Crypto API. The most direct and explicit method is using crypto.subtle.generateKey with the extractable flag set to true. For converting existing raw private key bytes into an extractable CryptoKey, @solana/keys provides createPrivateKeyFromBytes(bytes, true). Higher-level libraries like gill offer abstractions such as generateExtractableKeyPairSigner() for convenience, though they come with strong security caveats.

The transition of Solana's JavaScript SDKs towards Web Crypto API primitives like CryptoKeyPair signifies a positive alignment with web standards. This brings benefits such as improved browser compatibility, the use of standardized and audited cryptographic functions, and more granular control over key properties like extractability. However, it also necessitates a deeper understanding of these APIs and careful management of the associated security implications, especially when dealing with extractable private keys. The power to export key material must always be matched with an unwavering commitment to its secure handling to protect user assets and maintain the integrity of Solana accounts.

Works cited

1. Cryptography and the Solana Network, accessed May 6, 2025, https://solana.com/developers/courses/intro-to-solana/intro-to-cryptography
2. solana-com/content/courses/intro-to-solana/intro-to-cryptography.mdx at main - GitHub, accessed May 6, 2025, https://github.com/solana-foundation/solana-com/blob/main/content/courses/intro-to-solana/intro-to-cryptography.mdx
3. Class Keypair - solana/web3.js, accessed May 6, 2025, https://solana-labs.github.io/solana-web3.js/v1.x/classes/Keypair.html
4. @solana/keys - npm, accessed May 6, 2025, https://www.npmjs.com/package/%40solana%2Fkeys
5. SubtleCrypto: generateKey() method - Web APIs | MDN, accessed May 6, 2025, https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/generateKey
6. @solana/webcrypto-ed25519-polyfill - npm, accessed May 6, 2025, https://www.npmjs.com/package/%40solana%2Fwebcrypto-ed25519-polyfill
7. @solana/signers - npm, accessed May 6, 2025, https://www.npmjs.com/package/@solana/signers
8. How to Create a Keypair | Solana, accessed May 6, 2025, https://solana.com/developers/cookbook/wallets/create-keypair
9. gill/README.md at master · solana-foundation/gill · GitHub, accessed May 6, 2025, https://github.com/solana-foundation/gill/blob/master/README.md
10. Solana blockchain. How can i generate private key? - Stack Overflow, accessed May 6, 2025, https://stackoverflow.com/questions/69647006/solana-blockchain-how-can-i-generate-private-key
11. How to generate, save, importing a keypair into phantom with solana/web3.js v2?,

accessed May 6, 2025,
https://solana.stackexchange.com/questions/17681/how-to-generate-save-importing-a-keypair-into-phantom-with-solana-web3-js-v2