Enabling Solana Users to Access EVM dApps on Neon EVM

White Paper

Andrey Falaleev[1], Grzegorz Gancarczyk[1], Oleg Sukharev[2], Miroslav Nedelchev[2]

[1]Authors (a@neonfoundation.io, grzegorz.gancarczyk@neonfoundation.io)
[2]Reviewers (os@neonfoundation.io, miroslav.nedelchev@neonfoundation.io)

September 26, 2024

Abstract

This white paper presents a comprehensive technical solution designed to enable Solana users to seamlessly access EVM decentralized applications (dApps) deployed on Neon EVM. Currently, Solana users face significant barriers due to the requirement of EVM signatures and the necessity of using EVM wallets, such as MetaMask, to interact with Neon EVM. These challenges lead to a complicated user experience, liquidity fragmentation, and security concerns related to asset transfers across platforms.

Our proposed solution introduces support for Solana signatures within Neon EVM, allowing Solana users to sign Neon transactions using their existing Solana wallets. By generating associated Neon accounts to abstract Solana addresses and implementing an on-chain mempool system, we facilitate the execution of scheduled Neon transactions without the need for EVM wallets or additional asset transfers.

Moreover, we address the complexities involved in interactions between Ethereum Virtual Machine (EVM) contracts and Solana programs. Solana's lack of support for transaction reverts and immediate state application poses significant challenges for executing complex EVM contracts that interact with Solana programs. To overcome these obstacles, we introduce the concept of a controlled tree of Neon transactions, enabling atomic and parallel execution with controlled state changes and result aggregation.

These innovations aim to simplify the user experience by allowing Solana users to interact with Neon EVM using familiar tools, reduce liquidity fragmentation by eliminating the need for asset transfers between platforms, and enhance interoperability between Solana and Neon EVM. By providing mechanisms to handle complex interaction scenarios, we extend the capabilities of dApps deployed on Neon EVM, fostering a more integrated and efficient blockchain ecosystem that benefits both users and developers.

Contents

1 Introduction

The blockchain ecosystem comprises multiple platforms, each offering unique features and capabilities. Solana and Ethereum are two prominent platforms that have garnered significant attention due to their high performance and robust decentralized application (dApp) ecosystems, respectively. Neon EVM serves as a bridge between these platforms by enabling EVM smart contracts to run on the Solana blockchain.

However, Solana users currently face substantial barriers when attempting to access EVM dApps on Neon EVM. These challenges stem primarily from differences in transaction signing methods and execution models. This paper outlines a technical solution to overcome these barriers, thereby enhancing interoperability between Solana and Neon EVM and enriching the user experience.

2 Executive Summary

This white paper introduces a comprehensive solution that allows Solana users to interact with EVM dApps deployed on Neon EVM using their Solana wallets. The key components of this solution include:

- **Solana Signature Support**: Accepting Solana's ed25519 signatures within Neon EVM transactions.
- **Associated Neon Accounts**: Generating associated Neon accounts for Solana users to abstract their Solana addresses within Neon EVM.
- **On-Chain Mempool**: Implementing an on-chain mempool for Neon proxies to execute scheduled Neon transactions.
- **Controlled Tree of Neon Transactions**: Introducing a mechanism to handle complex interaction scenarios between EVM contracts and Solana programs.
- **Intent-Based Execution**: Enabling conditional transaction execution based on predefined intents.

By addressing both the user experience challenges and the technical complexities of cross-platform interactions, this solution aims to create a seamless and efficient environment for users and developers alike.

3 Problem Description

3.1 Challenges for Solana Users

Currently, executing a transaction on Neon EVM requires an EVM signature using the secp256k1 elliptic curve. Users must employ an EVM wallet, such as MetaMask, to create this signature. This requirement presents several disadvantages for Solana users:

- **User Experience Complexity**: Solana users must perform additional steps and utilize unfamiliar tools, complicating the interaction with Neon EVM.
- **Liquidity Fragmentation**: Assets must be transferred to their EVM-associated account in Neon EVM, which is represented by their EVM wallet's externally owned account (EOA). While tools like NeonPass provide an alternative to traditional bridges, liquidity remains fragmented at the Solana Virtual Machine (SVM) application level rather than at the chain level. In theory, all liquidity resides on Solana, but it is not accessible from regular Solana accounts and programs, necessitating the transfer of liquidity to associated Neon accounts. This process leads to fragmented liquidity and increased transaction costs.
- **Security Concerns**: The inability to use a single balance across both platforms, especially when controlled by hardware wallets, raises security issues.

- **Additional Tooling Requirements**: Dependence on EVM wallets adds overhead and potential points of failure.

3.2 Complex Interactions Between EVM Contracts and Solana Programs

Integrating EVM contracts with Solana programs poses several technical challenges:

- **Lack of Revert Support**: Solana programs do not support transaction reverts, a feature widely used in EVM contracts for error handling.
- **Atomicity Issues**: Failure in a call to a Solana program results in the failure of the entire transaction, which is not always desirable.
- **Immediate State Changes**: Solana programs apply state changes immediately, preventing the postponement of state modifications in complex interaction scenarios.
- **Transaction Size Limitations**: Solana transactions have size constraints, limiting the ability to interact with multiple Solana programs within a single transaction.

These challenges hinder the execution of complex EVM contracts that interact with Solana programs, limiting the functionality and interoperability of dApps on Neon EVM.

4 Proposed Solution

4.1 Overview

To overcome the identified challenges, we propose a solution that encompasses the following key components:

- **Solana Signature Support within Neon EVM**: Allow Solana users to sign Neon transactions using their Solana wallets by accepting Solana's ed25519 signatures in place of Ethereum's secp256k1 signatures.
- **On-Chain Mempool**: Implement an on-chain mempool system to store and manage scheduled Neon transactions, allowing Neon proxies to execute them on behalf of Solana users.
- **Associated Neon Accounts**: Generate associated Neon accounts to abstract Solana addresses within Neon EVM, enabling seamless user identification and interaction.
- **Controlled Tree of Neon Transactions**: Introduce a mechanism to manage complex interaction scenarios between EVM contracts and Solana programs, enabling atomic and parallel execution of transactions with controlled state changes.
- **Intent-Based Execution**: Incorporate intent-based execution to allow conditional transaction execution based on predefined conditions.

This solution aims to simplify the user experience for Solana users, reduce liquidity fragmentation, and enhance interoperability between Solana and Neon EVM.

4.2 Solana Signature Support

By modifying Neon EVM to accept Solana signatures, we eliminate the need for Solana users to utilize EVM wallets. This involves:

- **Signature Validation Adaptation**: Adjusting the transaction validation process in Neon EVM to accept ed25519 signatures.
- **Custom Transaction Types**: Utilizing the transaction `type` property to define custom transaction types compatible with Solana signatures.

4.3 On-Chain Mempool

An on-chain mempool within Neon EVM allows for:

- **Scheduling Transactions**: Solana users can schedule Neon transactions directly from their Solana wallets.
- **Proxy Execution**: Neon proxies can detect and execute scheduled transactions, handling complexities such as account preparation and iterative execution.
- **State Management**: The mempool facilitates the controlled execution of transactions, maintaining the integrity and atomicity of state changes.

The on-chain mempool feature enables the submission of large Ethereum transactions that exceed the typical 1 KB limit of Solana transactions. This is achieved by storing only the transaction hash on-chain, allowing users to send the full transaction "off-chain" to a proxy RPC service. This capability significantly enhances flexibility and scalability, as it permits Solana users to interact with larger and more complex transactions within the Neon EVM ecosystem.

Additionally, users have the option to **cancel scheduled transactions** before the operator's proxy has started the execution of the very first sub-transaction. This feature provides users with greater control over their transactions and the ability to prevent unintended executions.

4.4 Associated Neon Accounts

Generating associated Neon accounts for Solana users involves:

- **Address Abstraction**: Deriving EVM addresses from Solana public keys using the Keccak-256 hash function.
- **Account Mapping**: Mapping Solana users to their associated Neon accounts, enabling transaction tracking and balance management.
- **Balance Utilization**: Allowing the use of existing balances on Neon EVM without requiring additional asset transfers. To remove liquidity fragmentation, when a Solana user makes transactions with their SPL Tokens to the Neon EVM chain, an approval is made from their Associated Token Accounts (ATAs) to their Neon Program Derived Addresses (PDAs). This technical detail ensures that assets are seamlessly integrated and controlled within the user's accounts, maintaining liquidity without unnecessary fragmentation.

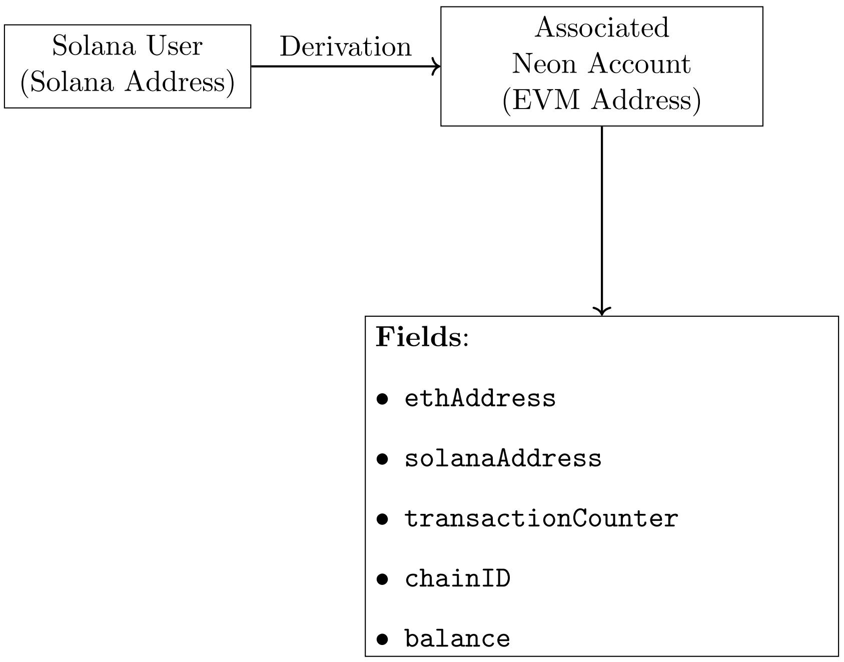

Figure 1: Associated Neon Account Structure

4.5 Controlled Tree of Neon Transactions

To handle complex interactions between EVM contracts and Solana programs, we introduce a controlled tree of Neon transactions:

- **Atomic Transactions**: Each Neon transaction is atomic, storing state changes upon finalization rather than at the end of the entire transaction tree.
- **Parallel Execution**: Transactions within the tree can be executed in parallel, controlled by the structure of the tree.
- **Result Aggregation**: Contracts can generate multiple transactions and aggregate results in subsequent transactions, allowing for complex interaction scenarios.

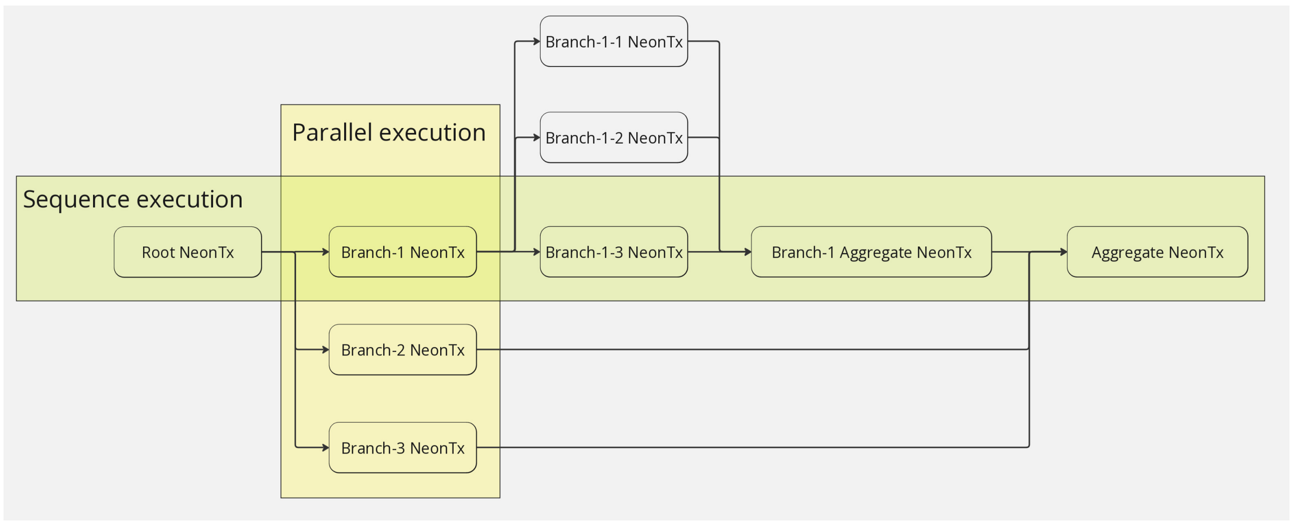

Figure 2: Process of a Tree Generation

The process of tree generation from the Root Neon Transaction involves the following steps:

1. A user initiates the process by sending a **Root NeonTx** to the Neon Proxy.

2. During the execution of the Root NeonTx, a contract generates three independent Neon Transactions: **Branch-1 NeonTx**, **Branch-2 NeonTx**, and **Branch-3 NeonTx**. The results from these transactions are then aggregated into the **Aggregate NeonTx**.
3. The **Branch-1 NeonTx**, **Branch-2 NeonTx**, and **Branch-3 NeonTx** can execute in parallel. The **Aggregate NeonTx** remains pending until all three branches finalize.
4. Within **Branch-1 NeonTx**, another contract generates three more independent transactions: **Branch-1-1 NeonTx**, **Branch-1-2 NeonTx**, and **Branch-1-3 NeonTx**. These transactions must complete before executing the **Branch-1 Aggregate NeonTx**.
    - Transactions **Branch-2 NeonTx**, **Branch-3 NeonTx**, **Branch-1-1 NeonTx**, **Branch-1-2 NeonTx**, and **Branch-1-3 NeonTx** are eligible for parallel execution.
    - The transactions **Branch-1-1 NeonTx**, **Branch-1-2 NeonTx**, and **Branch-1-3 NeonTx** can only begin after **Branch-1 NeonTx** has finalized.
5. Upon finalizing **Branch-1-1 NeonTx**, **Branch-1-2 NeonTx**, and **Branch-1-3 NeonTx**, the **Branch-1 Aggregate NeonTx** is executed.
6. Finally, once all the preceding transactions are complete, the final **Aggregate NeonTx** is executed, aggregating the results of all the branches.

4.6 Intent-Based Execution

Intent-based execution allows transactions to be executed only when certain conditions are met. This involves:

- **Conditional Execution**: Transactions include intents, which are simple EVM codes that define execution conditions.
- **Static Calls**: Intents must adhere to `staticcall` rules, meaning they cannot modify state or allocate storage.
- **Execution Control**: If conditions are not satisfied, the transaction is not executed, and the user does not incur costs.

This mechanism enhances control over transaction execution and resource utilization.

5 Advantages

The proposed solution offers several significant advantages:

- **Seamless User Experience**: Solana users can interact with EVM dApps using familiar tools without additional steps or wallets.
- **Liquidity Consolidation**: Eliminates the need for asset transfers and reduces liquidity fragmentation by allowing the use of a single balance across both Solana and Neon EVM.
- **Enhanced Security**: Reduces security risks associated with bridging assets between platforms, as users retain control over their assets within their Solana wallets.
- **Improved Interoperability**: Enables complex interactions between EVM contracts and Solana programs, expanding dApp capabilities and fostering innovation.
- **Optimized Performance**: Allows for parallel execution of transactions and efficient handling of state changes, improving overall system performance.
- **Developer Flexibility**: Provides developers with tools to build more integrated and complex applications, leveraging the strengths of both Solana and Ethereum ecosystems.

6 Technical Implementation

6.1 System Architecture

6.1.1 Architecture Overview

The system architecture integrates Solana users, Neon EVM, Neon proxies, and associated accounts to facilitate seamless interaction with EVM dApps.

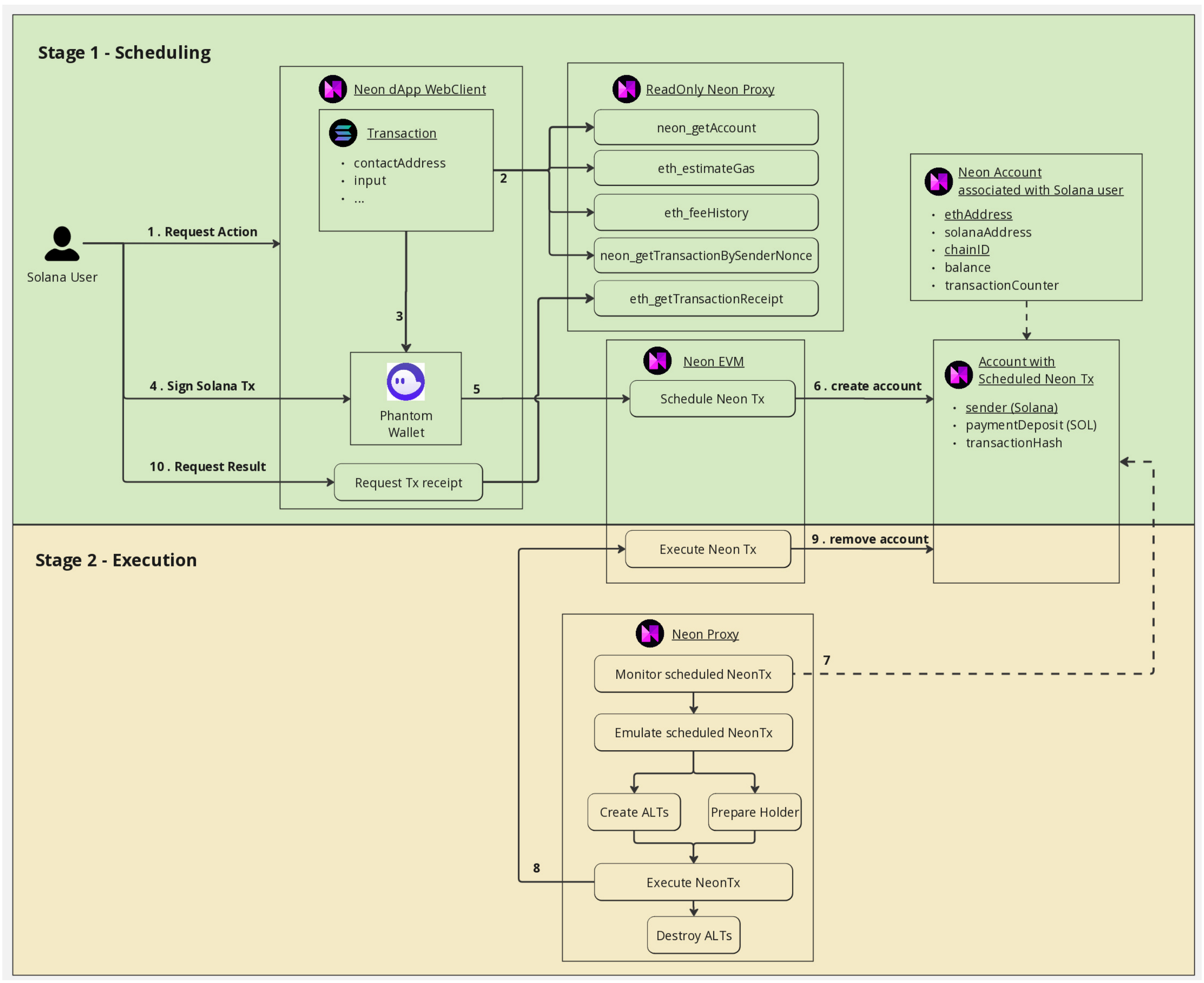

Figure 3: System Architecture of Neon EVM Integration with Solana Users

6.1.2 Components Description

- **Solana User**: The end-user holding a Solana wallet who interacts with Neon dApps.
- **Neon dApp**: The decentralized application deployed on Neon EVM, accessible via standard web interfaces.
- **Solana Wallet**: A wallet like Phantom, used by the Solana user to sign transactions.
- **Neon EVM Program**: The program running on Solana that implements the EVM functionality.
- **Neon Proxy**: A service that monitors the on-chain mempool and executes scheduled Neon transactions.
- **Associated Neon Account**: An account that maps the Solana user's address to an EVM address.

- **Solana Cluster**: The Solana blockchain network where transactions are processed.

6.2 Workflow for Scheduling Neon Transactions

The process for Solana users to schedule Neon transactions involves the following steps:

1. **Action Initiation**: The Solana user selects an action on the Neon dApp web interface.
2. **Preparation via RPC Calls**: The Neon dApp communicates with the Neon RPC endpoint to retrieve necessary account details and gas information.
3. **Transaction Signing**: The Neon dApp launches the Solana wallet to sign a Solana transaction containing the packed Neon transaction.
4. **Transaction Submission**: The Solana wallet submits the transaction to the Solana cluster for execution.
5. **On-Chain Mempool Storage**: The Solana cluster processes the transaction, and the Neon EVM program creates a TreeAccount and deposits funds from the Solana user for transaction execution.
6. **Neon Proxy Execution**: Neon proxies detect scheduled Neon transactions on the Solana blockchain and execute them, handling complexities such as emulation and preparation of Address Lookup Tables (ALTs).
7. **Result Monitoring**: The Neon dApp periodically requests the transaction status from the Neon RPC endpoint and displays progress to the user.

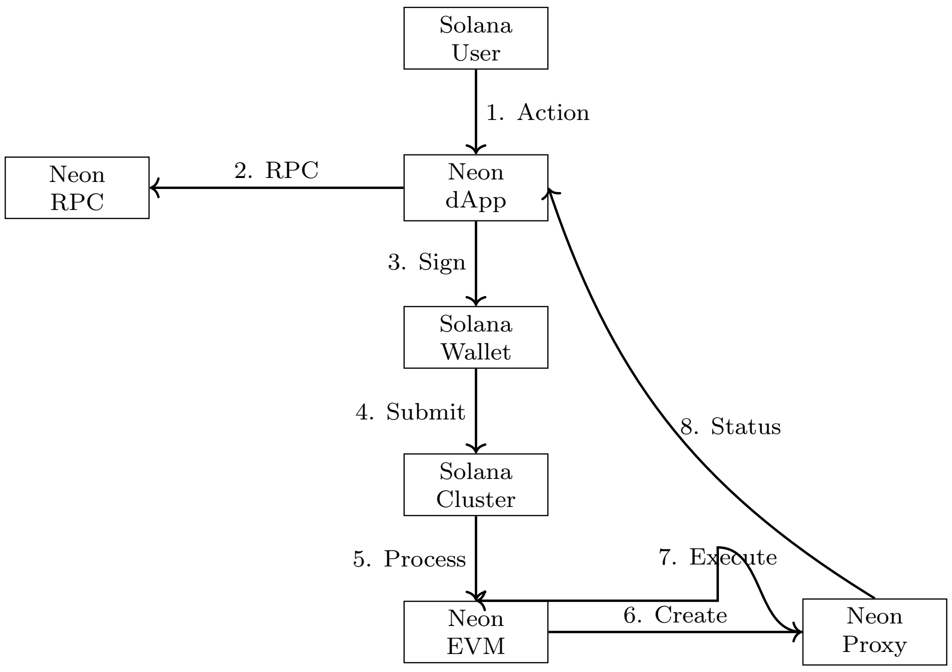

Figure 4: Workflow for Scheduling and Executing Neon Transactions by Solana Users

6.3 Solana Transaction Structure

A Solana transaction for scheduling Neon transactions includes:

- **Account List**:
  1. Solana user account.

2. Associated Neon account.
3. TreasuryAccount for funding the creation of the TreeAccount.
4. TreeAccount, a Program Derived Address (PDA) based on the sender and nonce.

- **Programs**:
  1. Neon EVM Program.
  2. Compute Budget Program.
- **Instructions**:
  1. Compute budget adjustments (set compute unit limits and prices).
  2. Neon EVM instruction containing the packed scheduled Neon transaction.

6.4 Variants of Neon Transaction Packing

There are two variants of how the Neon transaction can be packed into the Solana instruction:

1. **Full Scheduled Neon Transaction Body**:
   - The entire Neon transaction body is packed in Recursive Length Prefix (RLP) format.
   - This includes all transaction fields as defined in the scheduled Neon transaction body.
2. **List of Transaction Hashes**:
   - The instruction includes the following parameters:
     - `nonce`: Required to validate the address of the TreeAccount.
     - `chainID`: Defines the token space for gas payments.
     - `maxFeePerGas`: Required to calculate the execution balance.
     - `maxPriorityFeePerGas`: Required to calculate the execution balance.
     - `transactionHashList`: Up to 13 transaction hashes, each with associated parameters like `gasLimit`, `childTransactionIndex`, `successExecuteLimit`, and `transactionHash`.
   - This approach minimizes the size of the instruction data by referencing transaction hashes.

6.5 Execution of Scheduled Neon Transactions

Execution involves an additional stage for depositing SOLs from the Solana user's native balance to Neon EVM for gas payments:

1. **SOL Deposit**: SOLs are deposited into the TreeAccount's balance, reserved exclusively for transaction execution.
2. **Balance Utilization**: The associated Neon account's balance is utilized first before using the Solana user's native SOL balance.
3. **Transaction Processing**: Neon proxies execute the transactions, handling complexities such as account preparation and state management.
4. **Result Handling**: Upon completion, the results are returned, and any remaining balances are managed according to the fee rules.

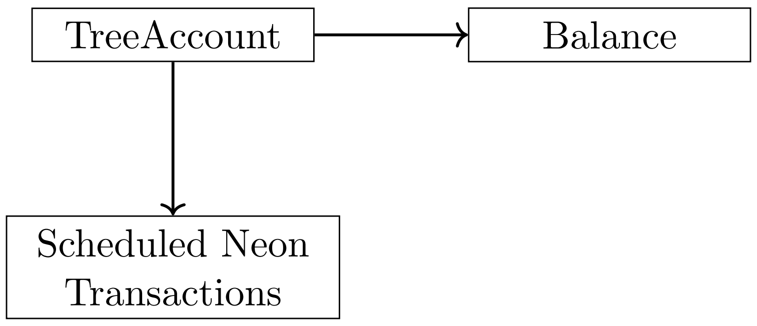

Figure 5: TreeAccount Structure

6.6 New Instructions in Neon EVM

Two new instructions are introduced:

1. **Schedule NeonTx**:
   - Validates that the TreeAccount does not already exist.
   - Creates the TreeAccount using the TreasuryAccount balance.
   - Deposits balance for Neon transaction execution from the associated Neon account and, if necessary, from the Solana user's native SOL balance.
2. **Deposit SOLs**:
   - Deposits SOLs to Neon EVM and stores wrapped SOLs (wSOL) in the TreeAccount balance.
   - Ensures that the native SOL balance is above the rent-exempt threshold.

6.7 Precompiled Contracts in Neon EVM

Three precompiled contracts are introduced to facilitate transaction scheduling and result retrieval:

1. **Create Local Tree of Scheduled Neon Transactions**:
   - Used by contracts to build a tree of scheduled transactions.
   - Input parameters include the number of transactions and their details.
2. **Create Aggregate NeonTx**:
   - Allows contracts to create an aggregate Neon transaction that depends on the completion of other transactions.
   - Manages dependencies and execution order.
3. **Request Execution Results**:
   - Contracts can request the results of executed parent transactions.
   - Ensures data integrity by verifying result hashes.

6.8 Order Execution from TreeAccount

The execution order is managed based on transaction dependencies and statuses:

- **Execution Conditions**:
  - A scheduled Neon transaction can start if its `parentCount` is zero and its `status` is `0xff` (not started).

  - If `successExecuteLimit` is greater than zero, the transaction may be skipped or have its limit decremented based on parent execution results.

- **Status Updates**:
  - Transactions update their status upon completion, influencing the execution of child transactions.
  - The `parentCount` of child transactions is decremented as parent transactions complete.

- **Parallel Execution**:
  - Transactions without dependencies can be executed in parallel.
  - Transactions sharing accounts may be executed sequentially to avoid conflicts.

7 Implementation Details

7.1 Expanded Technical Specifications

7.1.1 Data Structures

Scheduled Neon Transaction The scheduled Neon transaction includes the following fields:

- `type`: Custom transaction type identifier (`0x7f` for compatibility).
- `neonSubType`: Subtype to define scheduled transactions (`0x01`).
- `payer`: EVM address of the payer.
- `sender`: EVM address of the sender.
- `nonce`: Unique identifier for the transaction.
- `index`: Index within the transaction tree.
- `intentAddress`: Optional address specifying conditions for execution.
- `intentInput`: Data to pass into the intent.
- `contractAddress`: Address of the Neon contract to call.
- `contractInput`: Data to pass into the contract or creation code.
- `value`: Amount of native tokens to transfer.
- `chainID`: Chain namespace identifier.
- `gasLimit`: Maximum gas allowed for execution.
- `maxFeePerGas`: Maximum fee per gas unit.
- `maxPriorityFeePerGas`: Maximum priority fee per gas unit.

TreeAccount The TreeAccount structure includes:

- `type`: Identifier for Neon EVM.
- `payer`: EVM address of the payer.
- `lastSlot`: Last Solana slot in which the account was modified.
- `chainID`: Chain namespace identifier.
- `maxFeePerGas`: Maximum fee per gas unit.
- `maxPriorityFeePerGas`: Maximum priority fee per gas unit.
- `gasLimit`: Total gas limit for all scheduled Neon transactions.
- `balance`: Balance in native tokens.
- `lastIndex`: Auto-incremented value for unique transaction indexing.
- `transactionList`: List of scheduled Neon transactions with status and execution details.

7.1.2 Algorithms and Protocols

Address Derivation Algorithm The EVM address (`ethAddress`) for a Solana user is derived using the Keccak-256 hash function applied to the Solana public key (`solanaPublicKey`), similar to how Ethereum addresses are derived from Ethereum public keys:

$$\texttt{ethAddress} = \text{keccak256}(\texttt{solanaPublicKey})[12:32] \tag{1}$$

This method takes the last 20 bytes of the hash to form the address, consistent with Ethereum's approach where the address is the right-most 160 bits of the Keccak-256 hash of the ECDSA public key. Since the Solana public key is unique and the Keccak-256 hash function is collision-resistant, there is no concern of address collisions. Furthermore, the balance account includes the Solana user's address for verification, ensuring that only the owner of the corresponding Solana private key can authorize transactions for this account on Neon EVM.

Transaction Scheduling Protocol

1. The Neon EVM program receives a scheduling request.
2. It validates the transaction parameters and signatures.
3. A TreeAccount is created if it does not exist, using the `payer` and `nonce` as seeds.
4. The scheduled Neon transaction is added to the `transactionList` in the TreeAccount.
5. The `balance` is updated with deposited SOLs converted to wSOL.

7.2 Additional Details on RPC Methods

Extended RPC methods support interaction with scheduled Neon transactions:

- `neon_getTransactionBySenderNonce`:
  - Retrieves the transaction by sender and nonce.
  - Can include an optional `index` to specify scheduled transactions.
- `neon_getTransactionReceipt`:
  - Provides the execution result of a transaction.

  - Includes fields like `parentTransactionHash` and `childTransactions`.

- `neon_getScheduledTransactions`:
  - Returns the list of active scheduled transactions for a given sender.
  - Helps users track pending transactions.

7.3 Security Considerations

7.3.1 Signature Validation Mechanisms

The system ensures security by:

- **Signature Verification**: Solana signatures (ed25519) are verified using Solana's native cryptographic libraries.
- **Address Binding**: The derived `ethAddress` is uniquely associated with the `solanaPublicKey`, ensuring that only the Solana private key corresponding to the `solanaPublicKey` can authorize transactions for this account on Neon EVM. The balance account includes the Solana user address for double-checking, preventing unauthorized access.
- **Replay Protection**: The use of nonces and unique transaction indexes prevents replay attacks.

7.3.2 Validation of Intent Contracts

To ensure intents do not introduce security risks:

- **State Modification Restrictions**: Intents must adhere to `staticcall` rules, meaning they cannot modify state or allocate storage.
- **Execution Limits**: Intents should be small and executable within a single Solana transaction to prevent excessive resource consumption.
- **No Scheduling Within Intents**: Intents cannot schedule additional Neon transactions, avoiding unintended execution flows.

7.3.3 Mitigation of Potential Attack Vectors

- **Unauthorized Access**: Only the Solana user with the corresponding private key can authorize transactions from their associated Neon account.
- **Double-Spending**: The system tracks transaction nonces and balances to prevent double-spending of funds.
- **Data Integrity**: Hashes of transaction data are stored and verified to ensure data integrity during execution.
- **Handling Revert Logic**: Transactions that fail do not affect the entire transaction tree and are managed according to their `status` and execution limits.

7.4 Fee Rules and Economic Model

The fee structure is designed to minimize user costs:

- **TreasuryAccount Usage**: TreeAccounts are temporary objects with rent-exempt balances provided by the TreasuryAccount.
- **Token Returns**: Remaining gas tokens are transferred back to the payer's balance upon TreeAccount destruction.

- **Destruction Conditions**:
  - The account should not have active transaction processing.
  - The account can be destroyed if there are no unprocessed transactions and a specified time has passed since the last activity.
- **Fee Calculation**: Fees are calculated based on gas usage, with unused funds returned to the user.

8 Use Cases and Examples

8.1 Example: Solana User Interacting with a Neon dApp

Scenario A Solana user wants to swap tokens using a Neon dApp that requires token approval and swap execution.

Process

1. **Token Approval**: The user schedules a Neon transaction to approve the token transfer.
2. **Swap Execution**: The user schedules a second Neon transaction to execute the swap.
3. **Transaction Scheduling**: Both transactions are scheduled in a single Solana transaction signed by the Solana wallet.
4. **Neon Proxy Execution**: The Neon proxy detects the scheduled transactions and executes them in order.
5. **Result Aggregation**: The Neon dApp updates the user's balance based on the executed transactions.

Benefits

- The user performs both actions without needing an Ethereum wallet.
- Transactions are atomic and securely executed.
- The user's assets remain under their control throughout the process.

8.2 Example: Handling Solana Calls with Transaction Trees

Scenario An EVM developer wants to perform external calls to Solana programs and handle potential failures gracefully within their contract logic.

Process

1. **Isolating Solana Calls**: The developer splits the transaction into individual sub-transactions, each handling a separate call to a Solana program.
2. **Scheduling Sub-Transactions**: These sub-transactions are scheduled using the controlled tree mechanism.
3. **Execution and Failure Handling**: If a Solana call fails, only the corresponding sub-transaction fails, allowing the developer's contract to catch and handle the failure without affecting the entire transaction tree.
4. **Continuation of Logic**: The contract can proceed with subsequent logic, leveraging the controlled execution flow provided by the transaction tree.

Benefits

- Enables fine-grained error handling for Solana calls within EVM contracts.
- Improves robustness and reliability of cross-chain interactions.
- Enhances user experience by preventing entire transaction failures due to a single failed Solana call.

8.3 Example: Intent-Based Execution Scenario

Scenario A transaction should only be executed if a certain market condition is met (e.g., asset price reaches a threshold).

Process

1. **Intent Definition**: The user includes an intent in the scheduled Neon transaction that checks the market condition.
2. **Conditional Execution**: The Neon proxy evaluates the intent before execution.
3. **Execution Decision**:
   - If the condition is met, the transaction is executed.
   - If not, the transaction is skipped without cost to the user.

Benefits

- Users can automate transactions based on conditions.
- Prevents unnecessary execution and costs.
- Enhances flexibility in transaction management.

9 Changes and Updates

9.1 Neon EVM Program

Implementations include:

- **New Instructions**: Adding instructions for scheduling, starting, finalizing, and destroying Neon transactions and TreeAccounts.
- **Precompiled Contracts**: Introducing precompiled contracts to support transaction scheduling and result retrieval.
- **Transaction Origin Replacement**: Replacing `tx.origin` with the payer from the scheduled Neon transaction to maintain consistency.

9.2 Neon Core API

Enhancements involve:

- **Associated Account Retrieval**: Providing API methods to retrieve associated Neon accounts by Solana user addresses.
- **Gas Estimation Updates**: Including scheduled Neon transactions in gas estimation rules to improve accuracy.
- **RPC Methods Extension**: Extending methods to support interaction with scheduled transactions.

9.3 Neon Proxy

Updates include:

- **Extended RPC Methods**: Supporting scheduled transactions in methods like `neon_getTransactionBySen` and `neon_getTransactionReceipt`.
- **Monitoring and Execution**: Enhancing the Neon proxy to monitor scheduled transactions and manage their execution efficiently.
- **Instruction Parsing**: Updating parsing mechanisms to handle new Neon EVM instructions and transaction types.

10 Conclusion

By introducing Solana signature support, associated Neon accounts, and an on-chain mempool, we provide a solution that enables Solana users to access EVM dApps on Neon EVM seamlessly. Additionally, the ability to execute complex interaction scenarios between EVM contracts and Solana programs through a controlled tree of Neon transactions enhances the functionality and interoperability of the platform.

These innovations aim to simplify the user experience, reduce liquidity fragmentation, and promote broader adoption of Neon EVM among Solana users. Developers are empowered to build more complex and integrated applications, leveraging the strengths of both Solana and Ethereum ecosystems.

References

- Paradigm Research: Intent-Based Architecture
- EIP-2718: Typed Transaction Envelope
- Neon Labs Official Website
- Solana Documentation
- Neon Labs GitHub Repositories

A Appendix A: Glossary

ALT	Address Lookup Table, a mechanism to manage large lists of accounts in Solana transactions.
EVM	Ethereum Virtual Machine, the runtime environment for smart contracts in Ethereum.
Neon EVM	An implementation of the EVM on the Solana blockchain.
PDA	Program Derived Address, a deterministic address derived from a program and seeds in Solana.
Solana User	An individual or entity that holds a Solana wallet and interacts with the Solana blockchain.
TreeAccount	A specialized account in Neon EVM that stores a tree of scheduled Neon transactions.
wSOL	Wrapped SOL, an ERC-20 compatible version of SOL tokens.

B Appendix B: Technical Specifications

B.1 Keccak-256 Address Derivation

The EVM address for a Solana user is derived as follows:

$$\texttt{ethAddress} = \text{keccak256}(\texttt{solanaPublicKey})[12:32] \quad (2)$$

This takes the last 20 bytes of the hash to form the address.

B.2 Transaction Types

Custom transaction types are defined to maintain compatibility with Ethereum clients:

- **Type** = `0x7f`: Maximum value for Ethereum transaction types as per EIP-2718.
- **NeonSubType** = `0x01`: Defines a scheduled Neon transaction.

B.3 Gas Calculations

The gas calculations for rent-exempt accounts are as follows:

```
> solana rent 168
Rent-exempt minimum: 0.00206016 SOL

> solana rent 239
Rent-exempt minimum: 0.00255432 SOL

> echo $((0.00255432 - 0.00206016)) SOL
0.00049416 SOL
```

B.4 Fee Rules for TreeAccount

To minimize user payments:

- **TreasuryAccount Usage**: TreeAccounts are temporary objects with rent-exempt balances provided by the TreasuryAccount.
- **Token Returns**: Remaining gas tokens are transferred back to the payer's balance upon TreeAccount destruction.
- **Destruction Conditions**:
  - The account should not have active transaction processing.
  - The account can be destroyed if there are no unprocessed transactions and a specified time has passed since the last activity.

B.5 Example Transaction Data

Sample Scheduled Neon Transaction

- `type`: `0x7f`
- `neonSubType`: `0x01`
- `payer`: `0xabcdef...`
- `sender`: `0x123456...`
- `nonce`: `0x01`

- `index`: `0x00`
- `contractAddress`: `0xdeadbeef...`
- `contractInput`: `0xcafebabe...`
- `value`: `0`
- `gasLimit`: `21000`
- `maxFeePerGas`: `100 Gwei`
- `maxPriorityFeePerGas`: `2 Gwei`