Transactions

On the Solana blockchain, program execution begins with a transaction being submitted to the cluster. With each transaction consisting of one or many instructions , the runtime will process each of the instructions contained within the transaction, in order, and atomically. If any part of an instruction fails, then the entire transaction will fail.

Overview of a Transaction#

On Solana, clients update the runtime (for example, debiting an account) by submitting a transaction to the cluster.

This transaction consists of three parts:

- one or more instructions
- an array of accounts to read or write from
- one or more signatures

An instruction is the smallest execution logic on Solana. Instructions are basically a call to update the global Solana state. Instructions invoke programs that make calls to the Solana runtime to update the state (for example, calling the token program to transfer tokens from your account to another account).

Programs on Solana don't store data/state; rather, data/state is stored in accounts.

Signatures verify that we have the authority to read or write data to the accounts that we list.

Anatomy of a Transaction#

This section covers the binary format of a transaction.

Transaction Format#

A transaction contains a compact-array of signatures, followed by a message . Each item in the signatures array is a digital signature of the given message. The Solana runtime verifies that the number of signatures matches the number in the first 8 bits of the message header . It also verifies that each signature was signed by the private key corresponding to the public key at the same index in the message's account addresses array.

Signature Format#

Each digital signature is in the ed25519 binary format and consumes 64 bytes.

Message Format#

A message contains a header , followed by a compact-array of account addresses , followed by a recent blockhash , followed by a compact-array of instructions .

Message Header Format#

The message header contains three unsigned 8-bit values. The first value is the number of required signatures in the containing transaction. The second value is the number of those corresponding account addresses that are read-only. The third value in the message header is the number of read-only account addresses not requiring signatures.

Account Addresses Format#

The addresses that require signatures appear at the beginning of the account address array, with addresses requesting read-write access first, and read-only accounts following. The addresses that do not require signatures follow the addresses that do, again with read-write accounts first and read-only accounts following.

Blockhash Format#

A blockhash contains a 32-byte SHA-256 hash. It is used to indicate when a client last observed the ledger. Validators will reject transactions when the blockhash is too old.

Instruction Format#

An instruction contains a program id index, followed by a compact-array of account address indexes, followed by a compact-array of opaque 8-bit data. The program id index is used to identify an on-chain program that can interpret the opaque data. The program id index is an unsigned 8-bit index to an account address in the message's array of account addresses. The

account address indexes are each an unsigned 8-bit index into that same array.

Compact-Array Format#

A compact-array is serialized as the array length, followed by each array item. The array length is a special multi-byte encoding called compact-u16.

Compact-u16 Format#

A compact-u16 is a multi-byte encoding of 16 bits. The first byte contains the lower 7 bits of the value in its lower 7 bits. If the value is above 0x7f, the high bit is set and the next 7 bits of the value are placed into the lower 7 bits of a second byte. If the value is above 0x3fff, the high bit is set and the remaining 2 bits of the value are placed into the lower 2 bits of a third byte.

Account Address Format#

An account address is 32-bytes of arbitrary data. When the address requires a digital signature, the runtime interprets it as the public key of an ed25519 keypair.

Instructions#

Eachinstruction specifies a single program, a subset of the transaction's accounts that should be passed to the program, and a data byte array that is passed to the program. The program interprets the data array and operates on the accounts specified by the instructions. The program can return successfully, or with an error code. An error return causes the entire transaction to fail immediately.

Programs typically provide helper functions to construct instructions they support. For example, the system program provides the following Rust helper to construct aSystemInstruction::CreateAccount instruction:

pub fn create_account(from_pubkey: &Pubkey, to_pubkey: &Pubkey, lamports: u64, space: u64, owner: &Pubkey,) -> Instruction { let account_metas = vec![AccountMeta::new(*from_pubkey, true), AccountMeta::new(*to_pubkey, true),]; Instruction::new_with_bincode(system_program::id(), &SystemInstruction::CreateAccount { lamports, space, owner: *owner, }, account_metas,) }

Program Id#

The instruction'sprogram id specifies which program will process this instruction. The program's account's owner specifies which loader should be used to load and execute the program, and the data contains information about how the runtime should execute the program.

In the case ofon-chain SBF programs , the owner is the SBF Loader and the account data holds the BPF bytecode. Program accounts are permanently marked as executable by the loader once they are successfully deployed. The runtime will reject transactions that specify programs that are not executable.

Unlike on-chain programs,Native Programs are handled differently in that they are built directly into the Solana runtime.

Accounts#

The accounts referenced by an instruction represent on-chain state and serve as both the inputs and outputs of a program. More information about accounts can be found in theAccounts section.

Instruction data#

Each instruction carries a general purpose byte array that is passed to the program along with the accounts. The contents of the instruction data is program specific and typically used to convey what operations the program should perform, and any additional information those operations may need above and beyond what the accounts contain.

Programs are free to specify how information is encoded into the instruction data byte array. The choice of how data is encoded should consider the overhead of decoding, since that step is performed by the program on-chain. It's been observed that some common encodings (Rust's bincode for example) are very inefficient.

TheSolana Program Library's Token program gives one example of how instruction data can be encoded efficiently, but note that this method only supports fixed sized types. Token utilizes thePack trait to encode/decode instruction data for both token instructions as well as token account states.

Multiple instructions in a single transaction#

A transaction can contain instructions in any order. This means a malicious user could craft transactions that may pose

instructions in an order that the program has not been protected against. Programs should be hardened to properly and safely handle any possible instruction sequence.

One not so obvious example is account deinitialization. Some programs may attempt to deinitialize an account by setting its lamports to zero, with the assumption that the runtime will delete the account. This assumption may be valid between transactions, but it is not between instructions or cross-program invocations. To harden against this, the program should also explicitly zero out the account's data.

An example of where this could be a problem is if a token program, upon transferring the token out of an account, sets the account's lamports to zero, assuming it will be deleted by the runtime. If the program does not zero out the account's data, a malicious user could trail this instruction with another that transfers the tokens a second time.

Signatures#

Each transaction explicitly lists all account public keys referenced by the transaction's instructions. A subset of those public keys are each accompanied by a transaction signature. Those signatures signal on-chain programs that the account holder has authorized the transaction. Typically, the program uses the authorization to permit debiting the account or modifying its data. More information about how the authorization is communicated to a program can be found inAccounts

Recent Blockhash#

A transaction includes a recentblockhash to prevent duplication and to give transactions lifetimes. Any transaction that is completely identical to a previous one is rejected, so adding a newer blockhash allows multiple transactions to repeat the exact same action. Transactions also have lifetimes that are defined by the blockhash, as any transaction whose blockhash is too old will be rejected.