Native Programs in the Solana Runtime

Solana contains a small handful of native programs, which are required to run validator nodes. Unlike third-party programs, the native programs are part of the validator implementation and can be upgraded as part of cluster upgrades. Upgrades may occur to add features, fix bugs, or improve performance. Interface changes to individual instructions should rarely, if ever, occur. Instead, when change is needed, new instructions are added and previous ones are marked deprecated. Apps can upgrade on their own timeline without concern of breakages across upgrades.

For each native program the program id and description each supported instruction is provided. A transaction can mix and match instructions from different programs, as well include instructions from on-chain programs.

System Program

Create new accounts, allocate account data, assign accounts to owning programs, transfer lamports from System Program owned accounts and pay transaction fees.

- Program id:11111111111111111111111111111111
- Instructions:SystemInstruction

Config Program

Add configuration data to the chain and the list of public keys that are permitted to modify it

- Program id:Config1111111111111111111111111111111111111
- Instructions:config_instruction

Unlike the other programs, the Config program does not define any individual instructions. It has just one implicit instruction, a "store" instruction. Its instruction data is a set of keys that gate access to the account, and the data to store in it.

Stake Program

Create and manage accounts representing stake and rewards for delegations to validators.

- Program id:Stake11111111111111111111111111111111111111
- Instructions:StakeInstruction

Vote Program

Create and manage accounts that track validator voting state and rewards.

- Program id:Vote111111111111111111111111111111111111111
- Instructions:VoteInstruction

Address Lookup Table Program

- Program id:AddressLookupTab1e1111111111111111111111111
- Instructions:AddressLookupTableInstruction

BPF Loader

Deploys, upgrades, and executes programs on the chain.

- Program id:BPFLoaderUpgradeab1e11111111111111111111111
- Instructions:LoaderInstruction

The BPF Upgradeable Loader marks itself as "owner" of the executable and program-data accounts it creates to store your program. When a user invokes an instruction via a program id, the Solana runtime will load both your the program and its owner, the BPF Upgradeable Loader. The runtime then passes your program to the BPF Upgradeable Loader to process the instruction.

More information about deployment

Ed25519 Program

Verify ed25519 signature program. This program takes an ed25519 signature, public key, and message. Multiple signatures

can be verified. If any of the signatures fail to verify, an error is returned.

- Program id:Ed25519SigVerify111111111111111111111111111
- Instructions:new_ed25519_instruction

The ed25519 program processes an instruction. The firstu8 is a count of the number of signatures to check, which is followed by a single byte padding. After that, the following struct is serialized, one for each signature to check.

struct Ed25519SignatureOffsets { signature_offset: u16, // offset to ed25519 signature of 64 bytes signature_instruction_index: u16, // instruction index to find signature public_key_offset: u16, // offset to public key of 32 bytes public_key_instruction_index: u16, // instruction index to find public key message_data_offset: u16, // offset to start of message data message_data_size: u16, // size of message data message_instruction_index: u16, // index of instruction data to get message data } Pseudo code of the operation:

process_instruction() { for i in 0..count { // i'th index values referenced: instructions = &transaction.message().instructions instruction_index = ed25519_signature_instruction_index != u16::MAX ? ed25519_signature_instruction_index : current_instruction; signature = instructions[instruction_index].data[ed25519_signature_offset..ed25519_signature_offset + 64] instruction_index = ed25519_pubkey_instruction_index != u16::MAX ? ed25519_pubkey_instruction_index : current_instruction; pubkey = instructions[instruction_index].data[ed25519_pubkey_offset..ed25519_pubkey_offset + 32] instruction_index = ed25519_message_instruction_index != u16::MAX ? ed25519_message_instruction_index : current_instruction; message = instructions[instruction_index].data[ed25519_message_data_offset..ed25519_message_data_offset + ed25519_message_data_size] if pubkey.verify(signature, message) != Success { return Error } } return Success }

Secp256k1 Program

Verify secp256k1 public key recovery operations (ecrecover).

- Program id:KeccakSecp256k11111111111111111111111111111
- Instructions:new_secp256k1_instruction

The secp256k1 program processes an instruction which takes in as the first byte a count of the following struct serialized in the instruction data:

struct Secp256k1SignatureOffsets { secp_signature_offset: u16, // offset to [signature,recovery_id] of 64+1 bytes secp_signature_instruction_index: u8, // instruction index to find signature secp_pubkey_offset: u16, // offset to ethereum_address pubkey of 20 bytes secp_pubkey_instruction_index: u8, // instruction index to find pubkey secp_message_data_offset: u16, // offset to start of message data secp_message_data_size: u16, // size of message data secp_message_instruction_index: u8, // instruction index to find message data } Pseudo code of the operation:

process_instruction() { for i in 0..count { // i'th index values referenced: instructions = &transaction.message().instructions signature = instructions[secp_signature_instruction_index].data[secp_signature_offset..secp_signature_offset + 64] recovery_id = instructions[secp_signature_instruction_index].data[secp_signature_offset + 64] ref_eth_pubkey = instructions[secp_pubkey_instruction_index].data[secp_pubkey_offset..secp_pubkey_offset + 20] message_hash = keccak256(instructions[secp_message_instruction_index].data[secp_message_data_offset..secp_message_data_offset + secp_message_data_size]) pubkey = ecrecover(signature, recovery_id, message_hash) eth_pubkey = keccak256(pubkey[1..])[12..] if eth_pubkey != ref_eth_pubkey { return Error } } return Success } This allows the user to specify any instruction data in the transaction for signature and message data. By specifying a special instructions sysvar, one can also receive data from the transaction itself.

Cost of the transaction will count the number of signatures to verify multiplied by the signature cost verify multiplier.

Optimization notes

The operation will have to take place after (at least partial) deserialization, but all inputs come from the transaction data itself, this allows it to be relatively easy to execute in parallel to transaction processing and PoH verification. Previous Secure Vote Signing Next Runtime Sysvar Cluster Data