Cross-program Invocation (aka CPI)

The Solana runtime allows programs to call each other via a mechanism called "cross-program invocation ", orcpi for short. Calling between programs is achieved by one program invoking an instruction of the other. The invoking program is halted until the invoked program finishes processing the instruction.

For example, a client could create a transaction that modifies two accounts, each owned by separate on-chain programs:

let message = Message::new(vec![token_instruction::pay(&alice_pubkey), acme_instruction::launch_missiles(&bob_pubkey),]); client.send_and_confirm_message(&[&alice_keypair, &bob_keypair], &message); A client may instead allow theacme program to conveniently invoketoken instructions on the client's behalf:

let message = Message::new(vec![acme_instruction::pay_and_launch_missiles(&alice_pubkey, &bob_pubkey),]); client.send_and_confirm_message(&[&alice_keypair, &bob_keypair], &message); Given two on-chain programs,token andacme , each implementing instructionspay() andlaunch_missiles() respectively,acme can be implemented with a call to a function defined in thetoken module by issuing a cross-program invocation:

mod acme { use token_instruction;

```
fn launch_missiles(accounts: &[AccountInfo]) -> Result<()> {
    ...
}

fn pay_and_launch_missiles(accounts: &[AccountInfo]) -> Result<()> {
    let alice_pubkey = accounts[1].key;
    let instruction = token_instruction::pay(&alice_pubkey);
    invoke(&instruction, accounts)?;

    launch_missiles(accounts)?;
}
```

invoke() is built into Solana's runtime and is responsible for routing the given instruction to thetoken program via the instruction'sprogram_id field.

Note thatinvoke requires the caller to pass all the accounts required by the instruction being invoked, except for the executable account (theprogram_id).

Before invokingpay() , the runtime must ensure thatacme didn't modify any accounts owned bytoken . It does this by applying the runtime's policy to the current state of the accounts at the timeacme callsinvoke vs. the initial state of the accounts at the beginning of theacme 's instruction. Afterpay() completes, the runtime must again ensure thattoken didn't modify any accounts owned byacme by again applying the runtime's policy, but this time with thetoken program ID. Lastly, afterpay_and_launch_missiles() completes, the runtime must apply the runtime policy one more time where it normally would, but using all updatedpre_* variables. If executingpay_and_launch_missiles() up topay() made no invalid account changes,pay() made no invalid changes, and executing frompay() untilpay_and_launch_missiles() returns made no invalid changes, then the runtime can transitively assumepay_and_launch_missiles() as a whole made no invalid account changes, and therefore commit all these account modifications.

Instructions that require privileges#

The runtime uses the privileges granted to the caller program to determine what privileges can be extended to the callee. Privileges in this context refer to signers and writable accounts. For example, if the instruction the caller is processing contains a signer or writable account, then the caller can invoke an instruction that also contains that signer and/or writable account.

This privilege extension relies on the fact that programs are immutable, except during the special case of program upgrades.

In the case of theacme program, the runtime can safely treat the transaction's signature as a signature of atoken instruction. When the runtime sees thetoken instruction referencesalice_pubkey , it looks up the key in theacme instruction to see if that key corresponds to a signed account. In this case, it does and thereby authorizes thetoken program to modify Alice's account.

Program signed accounts#

Programs can issue instructions that contain signed accounts that were not signed in the original transaction by usingProgram derived addresses .

To sign an account with program derived addresses, a program mayinvoke_signed() .

invoke_signed(&instruction, accounts, &[&["First addresses seed"], &["Second addresses first seed", "Second addresses second seed"]],)?;

Call Depth#

Cross-program invocations allow programs to invoke other programs directly, but the depth is constrained currently to 4.

Reentrancy#

Reentrancy is currently limited to direct self recursion, capped at a fixed depth. This restriction prevents situations where a program might invoke another from an intermediary state without the knowledge that it might later be called back into. Direct recursion gives the program full control of its state at the point that it gets called back.

Program Derived Addresses#

Program derived addresses allow programmatically generated signatures to be used whencalling between programs .

Using a program derived address, a program may be given the authority over an account and later transfer that authority to another. This is possible because the program can act as the signer in the transaction that gives authority.

For example, if two users want to make a wager on the outcome of a game in Solana, they must each transfer their wager's assets to some intermediary that will honor their agreement. Currently, there is no way to implement this intermediary as a program in Solana because the intermediary program cannot transfer the assets to the winner.

This capability is necessary for many DeFi applications since they require assets to be transferred to an escrow agent until some event occurs that determines the new owner.

- Decentralized Exchanges that transfer assets between matching bid and ask
- orders.
- Auctions that transfer assets to the winner.
- Games or prediction markets that collect and redistribute prizes to the
- winners.

Program derived address:

1. Allow programs to control specific addresses, called program addresses, in
2. such a way that no external user can generate valid transactions with
3. signatures for those addresses.
4. Allow programs to programmatically sign for program addresses that are
5. present in instructions invoked viaCross-Program Invocations
6. .

Given the two conditions, users can securely transfer or assign the authority of on-chain assets to program addresses, and the program can then assign that authority elsewhere at its discretion.

Private keys for program addresses#

A program address does not lie on the ed25519 curve and therefore has no valid private key associated with it, and thus generating a signature for it is impossible. While it has no private key of its own, it can be used by a program to issue an instruction that includes the program address as a signer.

Hash-based generated program addresses#

Program addresses are deterministically derived from a collection of seeds and a program id using a 256-bit pre-image resistant hash function. Program address must not lie on the ed25519 curve to ensure there is no associated private key. During generation, an error will be returned if the address is found to lie on the curve. There is about a 50/50 chance of this happening for a given collection of seeds and program id. If this occurs a different set of seeds or a seed bump (additional 8 bit seed) can be used to find a valid program address off the curve.

Deterministic program addresses for programs follow a similar derivation path as Accounts created withSystemInstruction::CreateAccountWithSeed which is implemented withPubkey::create_with_seed .

For reference, that implementation is as follows:

pub fn create_with_seed(base: &Pubkey, seed: &str, program_id: &Pubkey,) -> Result { if seed.len() > MAX_ADDRESS_SEED_LEN { return Err(SystemError::MaxSeedLengthExceeded); }

```
Ok(Pubkey::new(
    hashv(&[base.as_ref(), seed.as_ref(), program_id.as_ref()]).as_ref(),
))
```

} Programs can deterministically derive any number of addresses by using seeds. These seeds can symbolically identify how the addresses are used.

FromPubkey ::

/// Generate a derived program address /// * seeds, symbolic keywords used to derive the key /// * program_id, program that the address is derived for pub fn create_program_address(seeds: &[&[u8]], program_id: &Pubkey,) -> Result

/// Find a valid off-curve derived program address and its bump seed /// * seeds, symbolic keywords used to derive the key /// * program_id, program that the address is derived for pub fn find_program_address(seeds: &[&[u8]], program_id: &Pubkey,) -> Option<(Pubkey, u8)> { let mut bump_seed = [std::u8::MAX]; for _ in 0..std::u8::MAX { let mut seeds_with_bump = seeds.to_vec(); seeds_with_bump.push(&bump_seed); if let Ok(address) = create_program_address(&seeds_with_bump, program_id) { return Some((address, bump_seed[0])); } bump_seed[0] -= 1; } None } Warning : Because of the way the seeds are hashed there is a potential for program address collisions for the same program id. The seeds are hashed sequentially which means that seedsabcdef,def, andefwill all result in the same program address given the same program id. Since the chance of collision is local to a given program id, the developer of that program must take care to choose seeds that do not collide with each other. For seed schemes that are susceptible to this type of hash collision, a common remedy is to insert separators between seeds, e.g. transformingdefintodef.

Using program addresses#

Clients can use thecreate_program_address function to generate a destination address. In this example, we assume thatcreate_program_address(&[&["escrow"]], &escrow_program_id) generates a valid program address that is off the curve.

// deterministically derive the escrow key let escrow_pubkey = create_program_address(&[&["escrow"]], &escrow_program_id);

// construct a transfer message using that key let message = Message::new(vec![token_instruction::transfer(&alice_pubkey, &escrow_pubkey, 1),]);

// process the message which transfer one 1 token to the escrow client.send_and_confirm_message(&[&alice_keypair], &message); Programs can use the same function to generate the same address. In the function below the program issues atoken_instruction::transfer from a program address as if it had the private key to sign the transaction.

fn transfer_one_token_from_escrow(program_id: &Pubkey, accounts: &[AccountInfo],) -> ProgramResult { // User supplies the destination let alice_pubkey = keyed_accounts[1].unsigned_key();

```
// Deterministically derive the escrow pubkey.
let escrow_pubkey = create_program_address(&[&["escrow"]], program_id);

// Create the transfer instruction
let instruction = token_instruction::transfer(&escrow_pubkey, &alice_pubkey, 1);

// The runtime deterministically derives the key from the currently
// executing program ID and the supplied keywords.
// If the derived address matches a key marked as signed in the instruction
// then that key is accepted as signed.
invoke_signed(&instruction, accounts, &[&["escrow"]])
```

} Note that the address generated usingcreate_program_address is not guaranteed to be a valid program address off the curve. For example, let's assume that the seed"escrow2" does not generate a valid program address.

To generate a valid program address using"escrow2" as a seed, usefind_program_address , iterating through possible bump seeds until a valid combination is found. The preceding example becomes:

// find the escrow key and valid bump seed let (escrow_pubkey2, escrow_bump_seed) = find_program_address(&[& ["escrow2"]], &escrow_program_id);

// construct a transfer message using that key let message = Message::new(vec![token_instruction::transfer(&alice_pubkey, &escrow_pubkey2, 1),]);

// process the message which transfer one 1 token to the escrow client.send_and_confirm_message(&[&alice_keypair], &message); Within the program, this becomes:

fn transfer_one_token_from_escrow2(program_id: &Pubkey, accounts: &[AccountInfo],) -> ProgramResult { // User supplies the destination let alice_pubkey = keyed_accounts[1].unsigned_key();

```
// Iteratively derive the escrow pubkey
let (escrow_pubkey2, bump_seed) = find_program_address(&[&["escrow2"]], program_id);

// Create the transfer instruction
let instruction = token_instruction::transfer(&escrow_pubkey2, &alice_pubkey, 1);

// Include the generated bump seed to the list of all seeds
invoke_signed(&instruction, accounts, &[&["escrow2", &[bump_seed]]])
```

} Sincefind_program_address requires iterating over a number of calls tocreate_program_address , it may use more[compute](#)

budget when used on-chain. To reduce the compute cost, usefind_program_address off-chain and pass the resulting bump seed to the program.

Instructions that require signers#

The addresses generated withcreate_program_address andfind_program_address are indistinguishable from any other public key. The only way for the runtime to verify that the address belongs to a program is for the program to supply the seeds used to generate the address.

The runtime will internally callcreate_program_address , and compare the result against the addresses supplied in the instruction.

Examples#

Refer toDeveloping with Rust andDeveloping with C for examples of how to use cross-program invocation.