FAQ

When writing or interacting with Solana programs, there are common questions or challenges that often come up. Below are resources to help answer these questions.

If not addressed here, ask onStackExchange with thesolana-program tag.

Limitations#

Developing programs on the Solana blockchain have some inherent limitation associated with them. Below is a list of common limitation that you may run into.

Seelimitations of developing programs for more details

Berkeley Packet Filter (BPF)#

Solana on-chain programs are compiled via theLLVM compiler infrastructure to anExecutable and Linkable Format (ELF) containing a variation of theBerkeley Packet Filter (BPF) bytecode.

Because Solana uses the LLVM compiler infrastructure, a program may be written in any programming language that can target the LLVM's BPF backend.

BPF provides an efficientinstruction set that can be executed in an interpreted virtual machine or as efficient just-in-time compiled native instructions.

Memory map#

The virtual address memory map used by Solana SBF programs is fixed and laid out as follows

- Program code starts at 0x100000000
- Stack data starts at 0x200000000
- Heap data starts at 0x300000000
- Program input parameters start at 0x400000000

The above virtual addresses are start addresses but programs are given access to a subset of the memory map. The program will panic if it attempts to read or write to a virtual address that it was not granted access to, and anAccessViolation error will be returned that contains the address and size of the attempted violation.

InvalidAccountData#

This program error can happen for a lot of reasons. Usually, it's caused by passing an account to the program that the program is not expecting, either in the wrong position in the instruction or an account not compatible with the instruction being executed.

An implementation of a program might also cause this error when performing a cross-program instruction and forgetting to provide the account for the program that you are calling.

InvalidInstructionData#

This program error can occur while trying to deserialize the instruction, check that the structure passed in matches exactly the instruction. There may be some padding between fields. If the program implements the RustPack trait then try packing and unpacking the instruction typeT to determine the exact encoding the program expects.

MissingRequiredSignature#

Some instructions require the account to be a signer; this error is returned if an account is expected to be signed but is not.

An implementation of a program might also cause this error when performing a cross-program invocation that requires a signed program address, but the passed signer seeds passed toinvoke_signed don't match the signer seeds used to create the program addresscreate_program_address .

rand Rust dependency causes compilation failure#

SeeRust Project Dependencies

Rust restrictions#

SeeRust restrictions

Stack#

SBF uses stack frames instead of a variable stack pointer. Each stack frame is 4KB in size.

If a program violates that stack frame size, the compiler will report the overrun as a warning.

For example:

Error: Function _ZN16curve25519_dalek7edwards21EdwardsBasepointTable6create17h178b3d2411f7f082E Stack offset of -30728 exceeded max offset of -4096 by 26632 bytes, please minimize large stack variables The message identifies which symbol is exceeding its stack frame, but the name might be mangled if it is a Rust or C++ symbol.

To demangle a Rust symbol userustfilt . The above warning came from a Rust program, so the demangled symbol name is:

rustfilt _ZN16curve25519_dalek7edwards21EdwardsBasepointTable6create17h178b3d2411f7f082E curve25519_dalek::edwards::EdwardsBasepointTable::create To demangle a C++ symbol usec++filt from binutils.

The reason a warning is reported rather than an error is because some dependent crates may include functionality that violates the stack frame restrictions even if the program doesn't use that functionality. If the program violates the stack size at runtime, anAccessViolation error will be reported.

SBF stack frames occupy a virtual address range starting at0x200000000 .

Heap size#

Programs have access to a runtime heap either directly in C or via the Rustalloc APIs. To facilitate fast allocations, a simple 32KB bump heap is utilized. The heap does not supportfree orrealloc so use it wisely.

Internally, programs have access to the 32KB memory region starting at virtual address 0x300000000 and may implement a custom heap based on the program's specific needs.

- Rust program heap usage
- C program heap usage

Loaders#

Programs are deployed with and executed by runtime loaders, currently there are two supported loaders BPF Loader andBPF loader deprecated

Loaders may support different application binary interfaces so developers must write their programs for and deploy them to the same loader. If a program written for one loader is deployed to a different one the result is usually aAccessViolation error due to mismatched deserialization of the program's input parameters.

For all practical purposes program should always be written to target the latest BPF loader and the latest loader is the default for the command-line interface and the javascript APIs.

For language specific information about implementing a program for a particular loader see:

- Rust program entrypoints
- C program entrypoints

Deployment#

SBF program deployment is the process of uploading a BPF shared object into a program account's data and marking the account executable. A client breaks the SBF shared object into smaller pieces and sends them as the instruction data ofWrite instructions to the loader where loader writes that data into the program's account data. Once all the pieces are received the client sends aFinalize instruction to the loader, the loader then validates that the SBF data is valid and marks the program account asexecutable . Once the program account is marked executable, subsequent transactions may issue instructions for that program to process.

When an instruction is directed at an executable SBF program the loader configures the program's execution environment, serializes the program's input parameters, calls the program's entrypoint, and reports any errors encountered.

For further information, seedeploying programs .

Input Parameter Serialization#

SBF loaders serialize the program input parameters into a byte array that is then passed to the program's entrypoint, where the program is responsible for deserializing it on-chain. One of the changes between the deprecated loader and the current loader is that the input parameters are serialized in a way that results in various parameters falling on aligned offsets within the aligned byte array. This allows deserialization implementations to directly reference the byte array and provide aligned pointers to the program.

For language specific information about serialization see:

- Rust program parameter deserialization
- C program parameter deserialization

The latest loader serializes the program input parameters as follows (all encoding is little endian):

- 8 bytes unsigned number of accounts
- For each account* 1 byte indicating if this is a duplicate account, if not a duplicate then
- - the value is 0xff, otherwise the value is the index of the account it is a
- - duplicate of.
- - If duplicate: 7 bytes of padding
- - If not duplicate:* 1 byte boolean, true if account is a signer
- - - 1 byte boolean, true if account is writable
- - - 1 byte boolean, true if account is executable
- - - 4 bytes of padding
- - - 32 bytes of the account public key
- - - 32 bytes of the account's owner public key
- - - 8 bytes unsigned number of lamports owned by the account
- - - 8 bytes unsigned number of bytes of account data
- - - x bytes of account data
- - - 10k bytes of padding, used for realloc
- - - enough padding to align the offset to 8 bytes.
- - - 8 bytes rent epoch
- 8 bytes of unsigned number of instruction data
- x bytes of instruction data
- 32 bytes of the program id