Reverse Engineering

Solana Programs with Radare2

An Introduction to the Solana runtime, the sBPF architecture
and the new sBPF plug-in in radare2

INVERSIVE LABS

Security Automation, Research and Engineering
for Solana and other Non-EVM Ecosystems

@seecoalba

inversive.xyz

@ulexec

Sergio Corrales

Co-Founder and Security Researcher

Ignacio Sanmillan

Co-Founder and Security Researcher

INVERSIVE LABS

Outline

1. An Introduction to Solana

2. An Overview of the Solana Runtime

3. The sBPF Architecture

4. Radare2 Support for Solana Programs

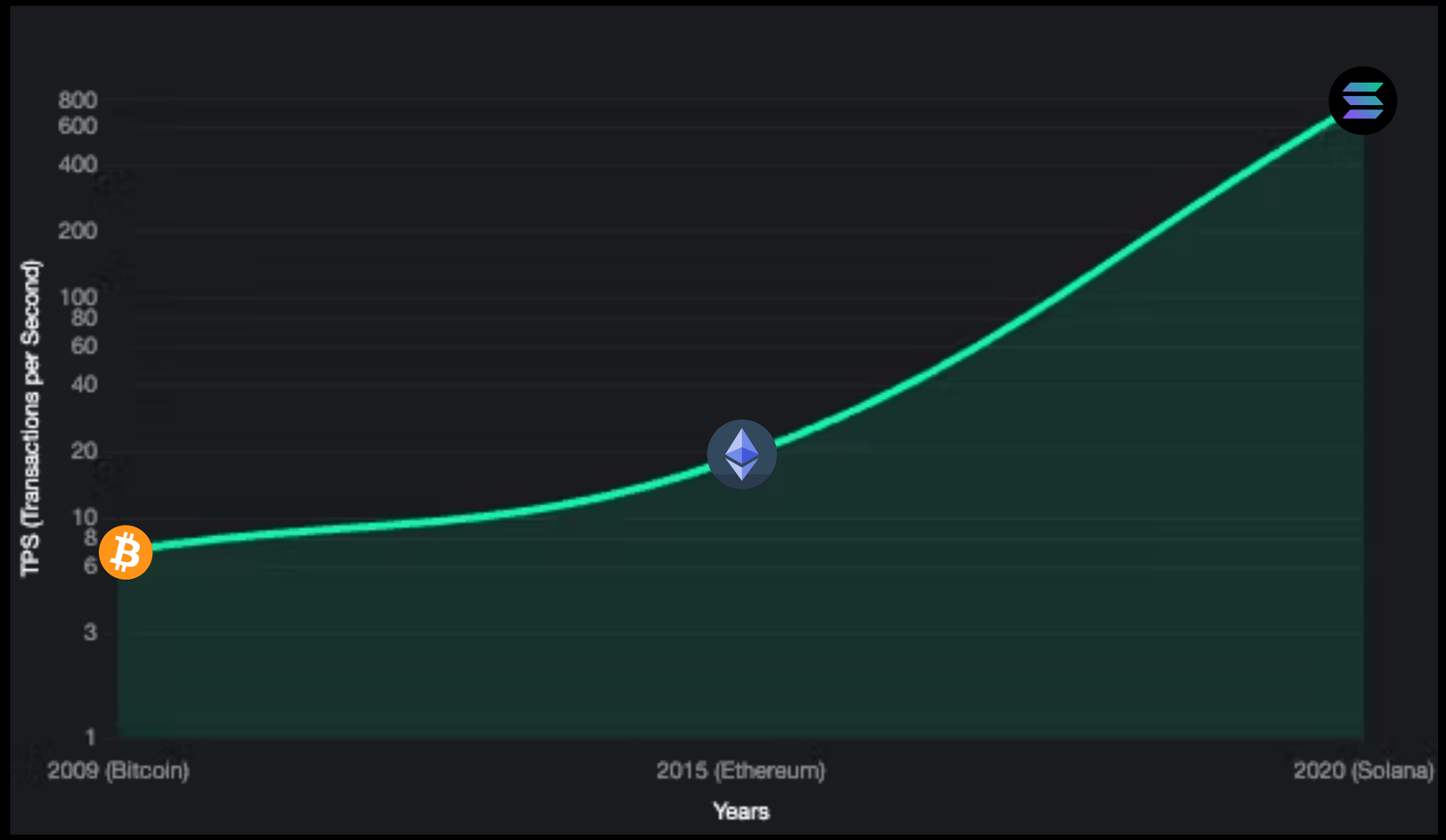
1. An Introduction to Solana

- Solana is a **high-performance** blockchain platform designed for **decentralized** applications and cryptocurrency-based transactions.

- It's designed to **scale** without sacrificing **decentralization** or security, making it popular for DeFi, NFTs, and high-frequency applications.

- Solana emerged to address scalability bottlenecks in other blockchains.
- Network congestion issues lead to high fees and slow confirmation times (up to 12 seconds per block and only ~15 TPS).
- To address this, it designed a new architecture from scratch, featuring stateless programs (no internal state) and separate data accounts, enabling parallel execution of non-conflicting transactions across multiple CPU cores.

October 2025



Bitcoin: ~7 TPS

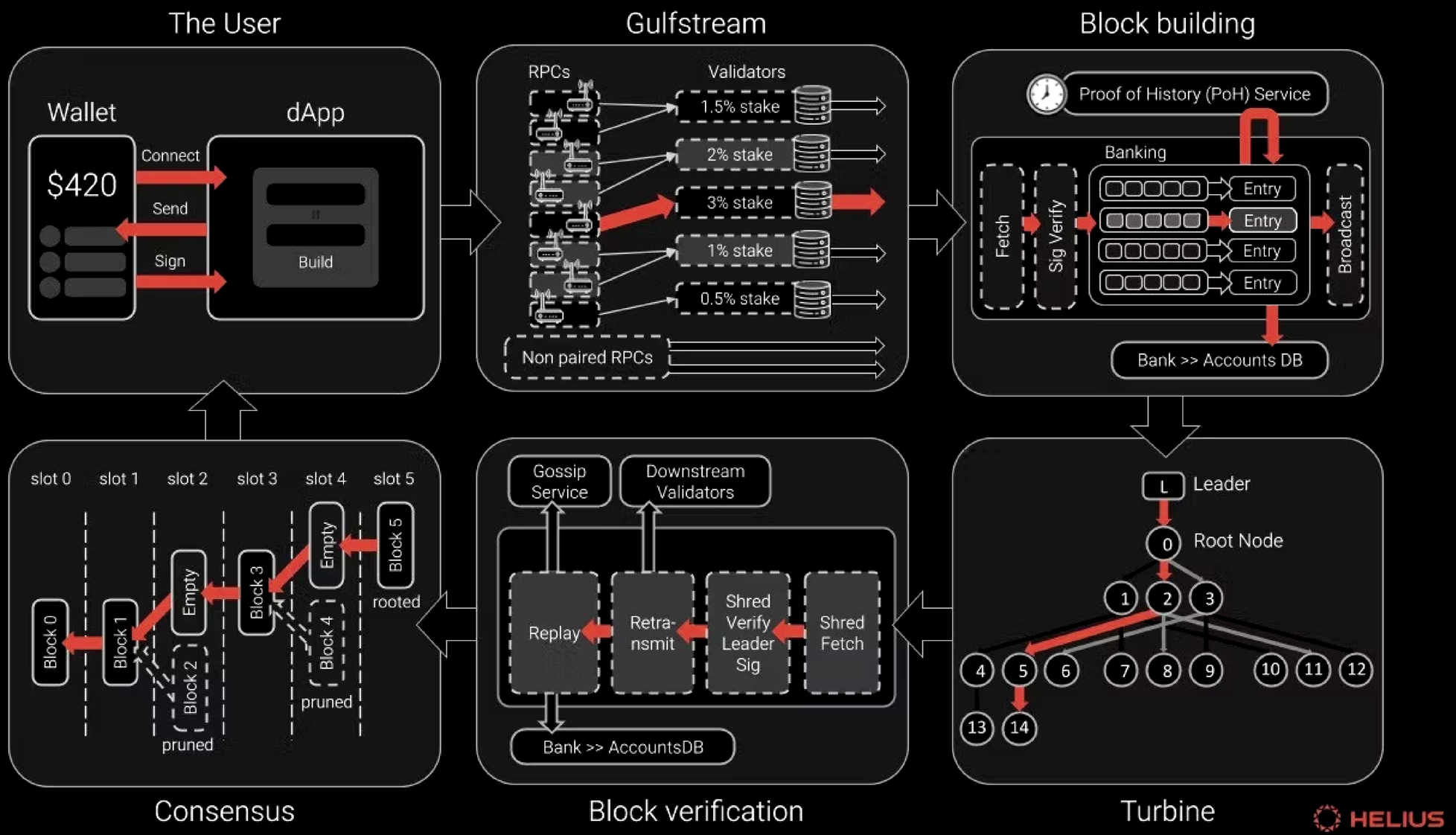
Ethereum: 19.45 TPS

Solana: 790.1 TPS

https://chainspect.app/dashboard

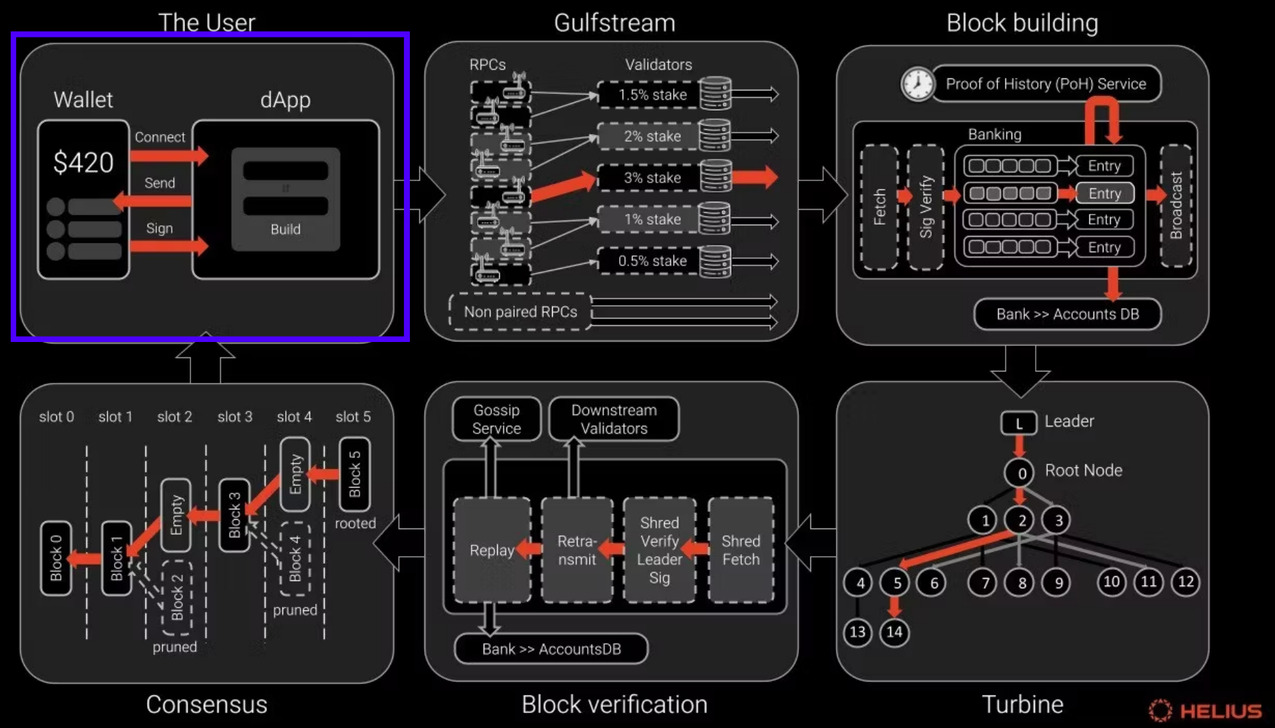
INVERSIVE LABS

- **Fast**: Processes ~65,000 transactions per second with sub-second block times
- **Low cost**: Transaction fees are typically fractions of a penny
- **Proof of History (PoH)**: Uses a cryptographic clock to order transactions before consensus, enabling high throughput
- **Proof of Stake (PoS)**: Uses validators to secure the network
- **Account-based model**: Data and programs are stored in accounts, not a traditional smart contract state
- **Rust/C programs**: Smart contracts (programs) are typically written in Rust and compiled to sBPF bytecode



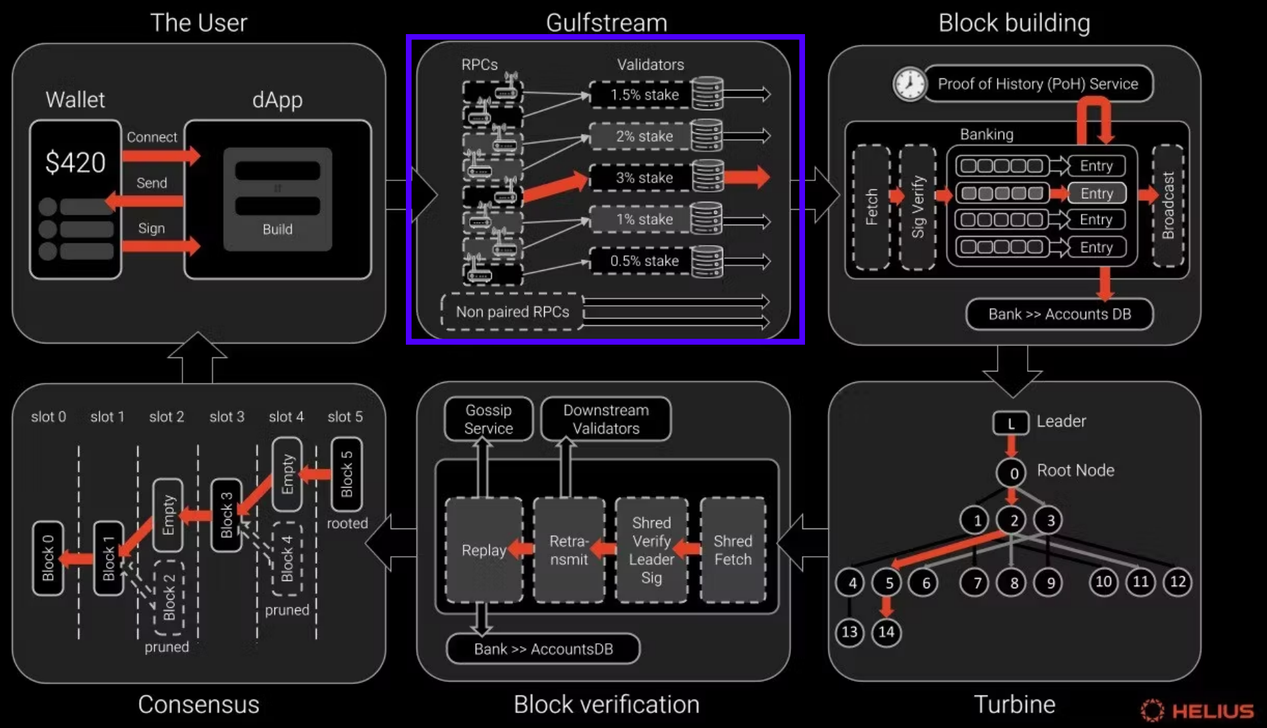
https://www.helius.dev/blog/solana-executive-overview

INVERSIVE LABS



- Transactions begin when users sign and submit them from wallets through dApps (decentralized applications), specifying actions like transfers with upfront account lists for secure, atomic processing.

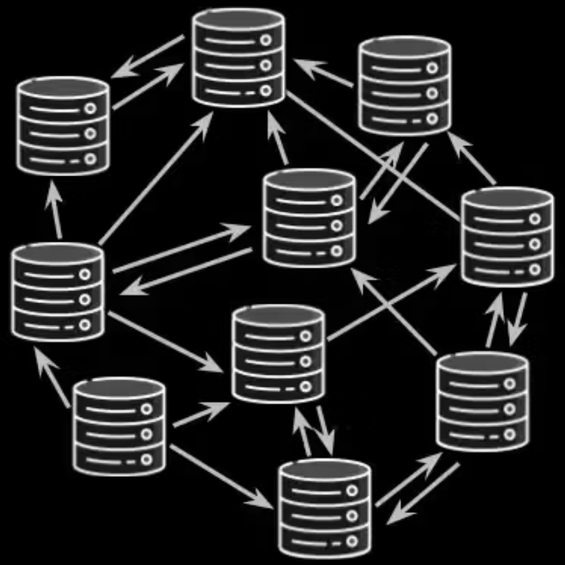
INVERSIVE LABS



- The term "Gulf Stream" can be loosely defined as the process that occurs from the moment a node picks up a transaction on the network until it reaches the leader for the current slot
- Transactions forward directly to the stake-weighted leader validator via RPC gateways, bypassing a mempool for fast, prioritized delivery

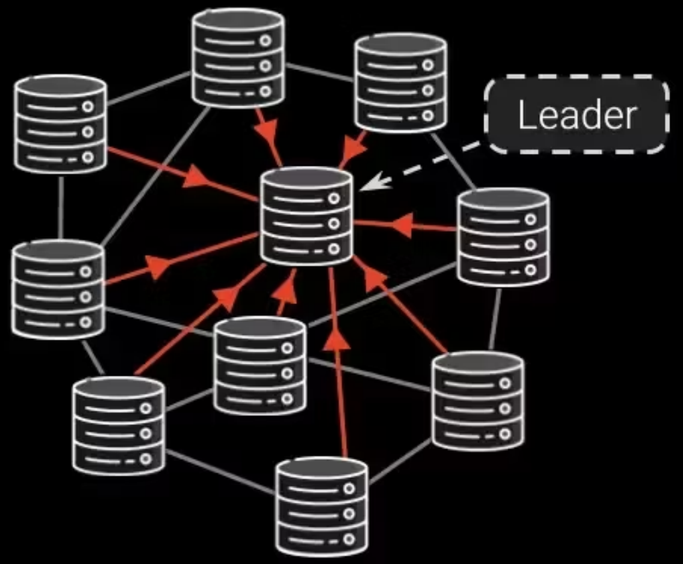
INVERSIVE LABS

Ethereum



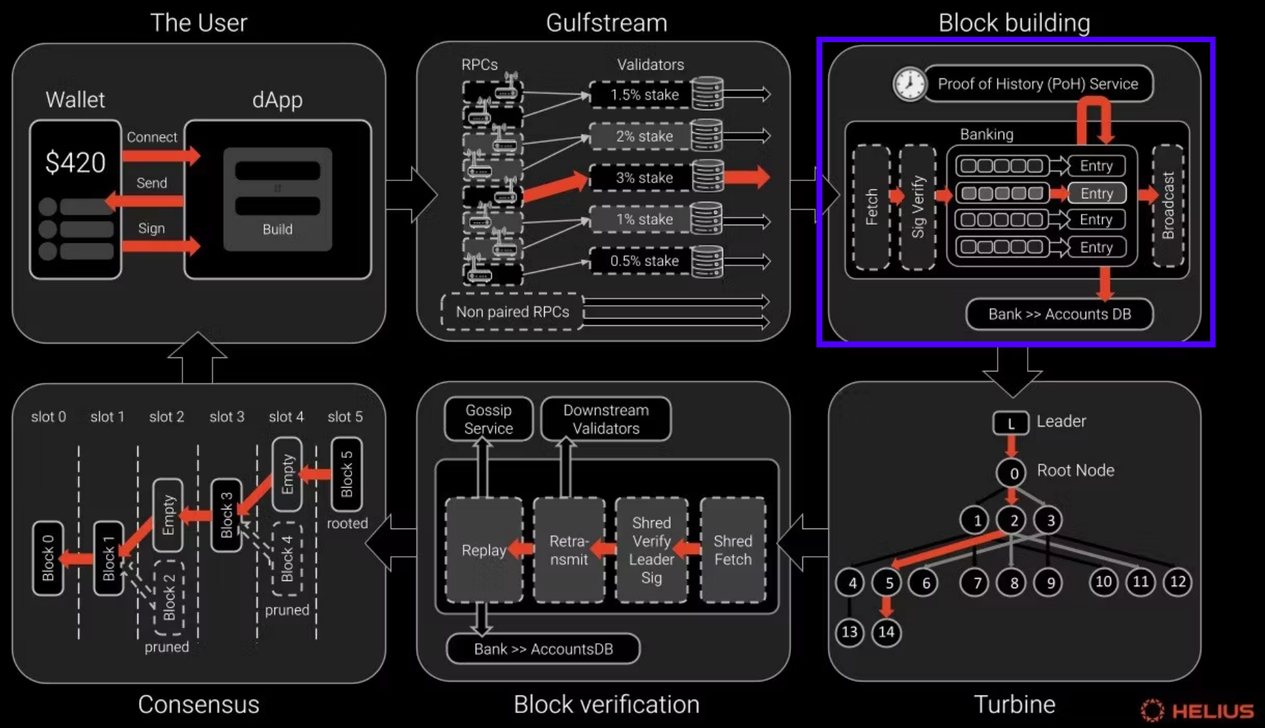
Gossip network + Mem-pool

Solana



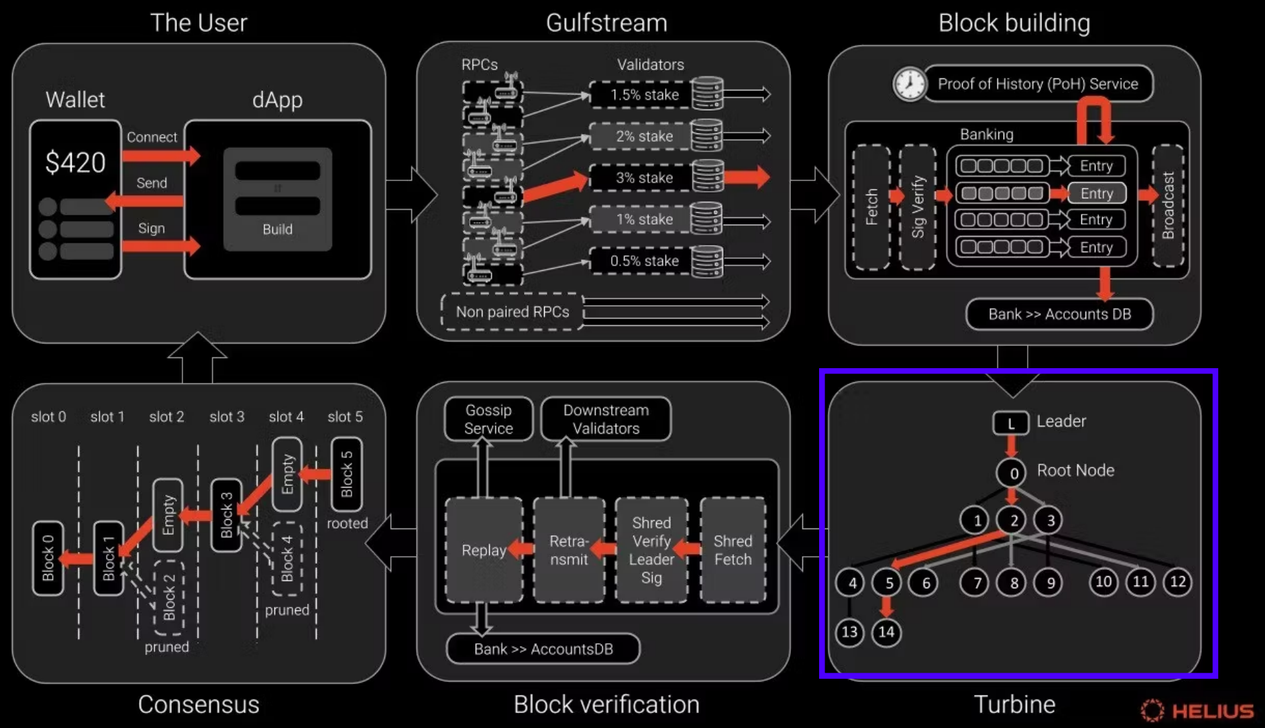
Gulf Stream + Leader schedule

INVERSIVE LABS



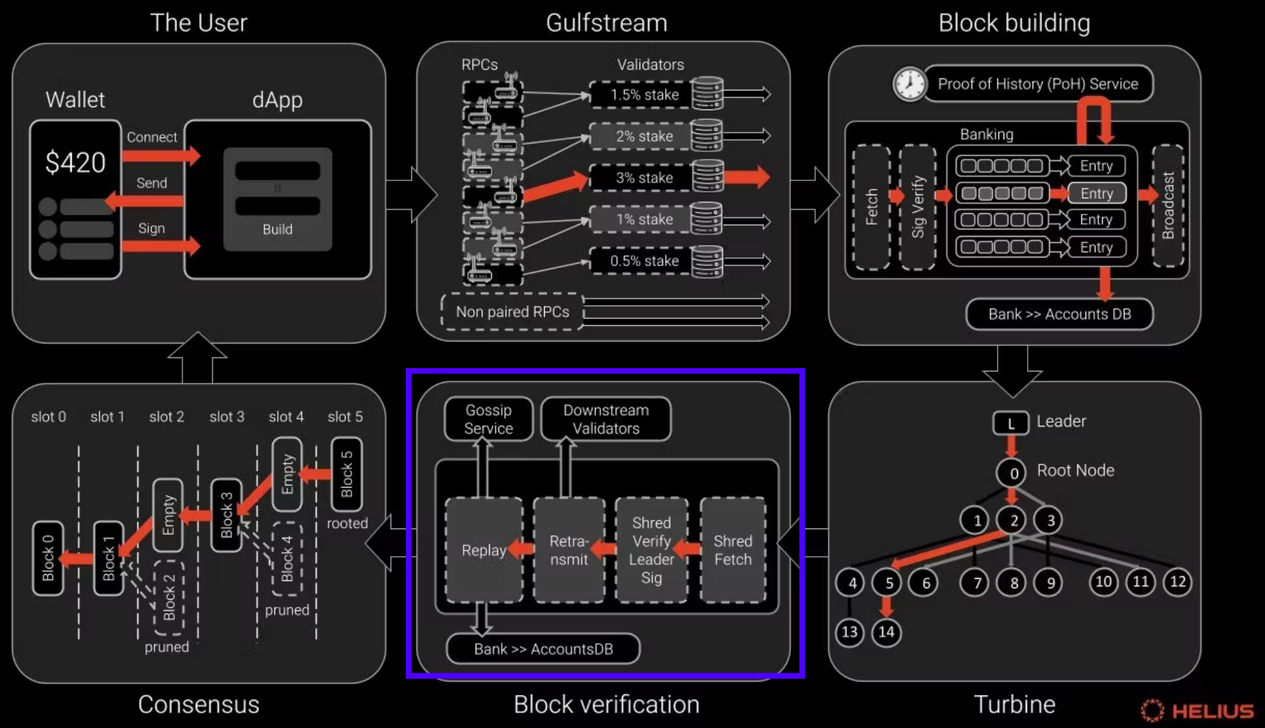
- The leader verifies signatures, groups non-conflicting transactions into bundles, executes them in parallel using Proof of History timestamps, and assembles a block over a time slot.

INVERSIVE LABS



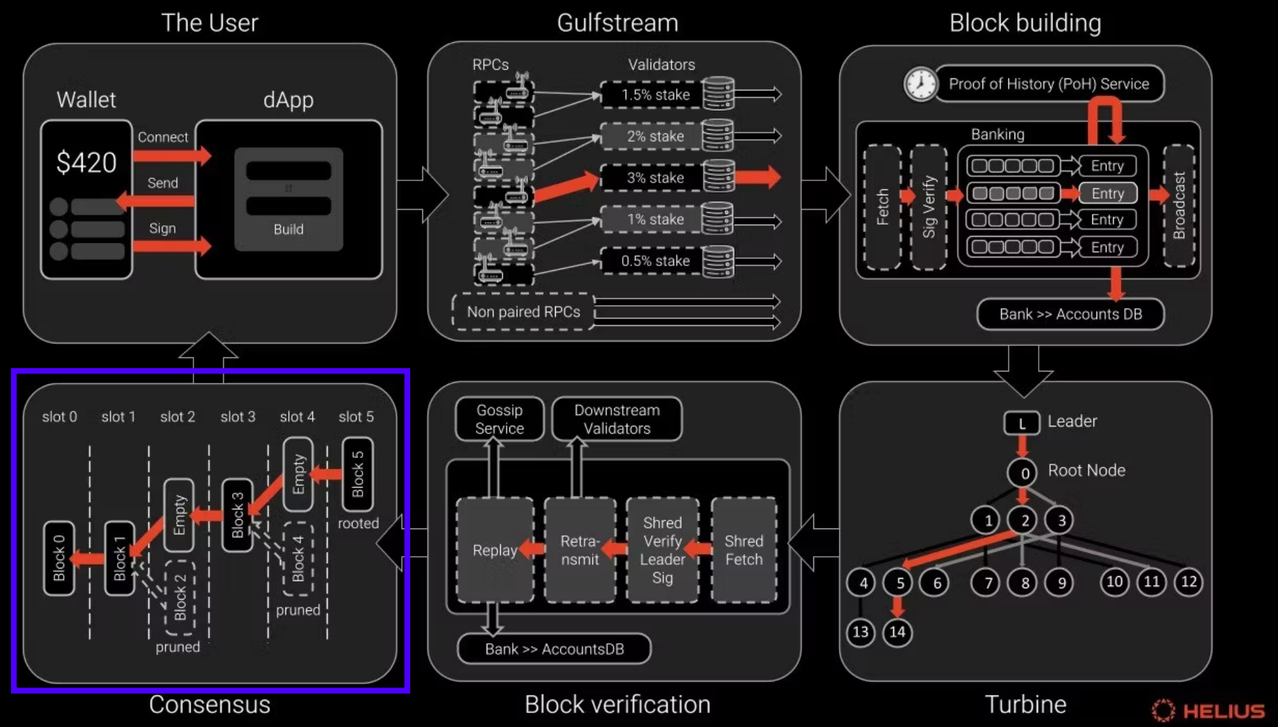
- Turbine is the process through which the leader [propagates their block](#) to the rest of the network
- Validators vote on blocks in a single round using Tower BFT, selecting the heaviest fork for quick confirmation and finality based on stake-weighted agreement.

INVERSIVE LABS



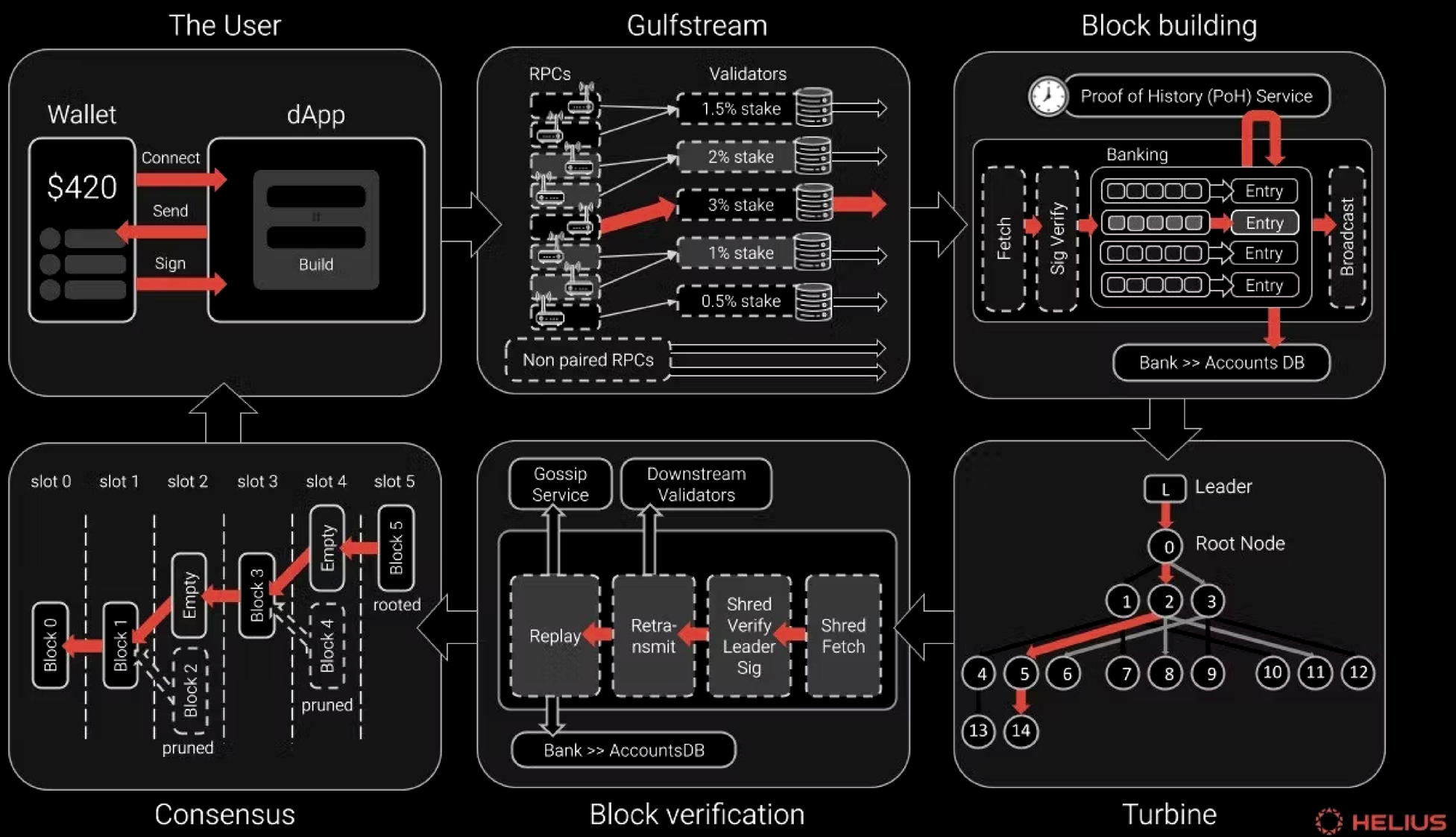
- Once a validator receives a new block from the leader via Turbine, it must validate all transactions within each entry.

INVERSIVE LABS

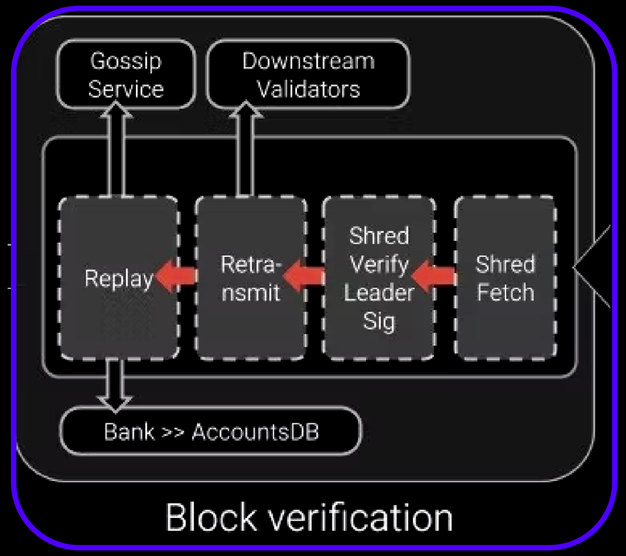


- Validators vote on blocks in a single round using Tower BFT, selecting the heaviest blockchain fork for quick confirmation and finality based on stake-weighted agreement.

INVERSIVE LABS


The User
Wallet
dApp
$420
Connect
Send
Sign
Build
Gulfstream
RPCs
Validators
1.5% stake
2% stake
3% stake
1% stake
0.5% stake
Non paired RPCs
Block building
Proof of History (PoH) Service
Banking
Fetch
Sig Verify
Entry
Entry
Entry
Entry
Broadcast
Bank >> Accounts DB
slot 0
slot 1
slot 2
slot 3
slot 4
slot 5
Block 0
Block 1
Empty
Block 2
pruned
Block 3
Empty
Block 4
pruned
Block 5
rooted
Consensus
Gossip Service
Downstream Validators
Replay
Retra-nsmit
Shred Verify Leader Sig
Shred Fetch
Bank >> AccountsDB
Block verification
L
Leader
0
Root Node
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Turbine
HELIUS
INVERSIVE LABS

In this presentation, we'll focus specifically on
key aspects of this component.



2. An Overview of the Solana Runtime

- The Solana Runtime is the core **execution environment** that processes **transactions** and **executes programs** on the Solana blockchain.

- The Solana Runtime is the core **execution environment** that processes **transactions** and **executes programs** on the Solana blockchain.

- It encompasses everything from **transaction validation** to **program execution**, acting as the bridge between the network layer and the on-chain state.

- The Solana Runtime is the core **execution environment** that processes **transactions** and **executes programs** on the Solana blockchain.

- It encompasses everything from **transaction validation** to **program execution**, acting as the bridge between the network layer and the on-chain state.

- When executing sBPF programs (solana smart contracts), the **BPF Loader** program handles the complexity of running untrusted code.

- The Solana Runtime is the core **execution environment** that processes **transactions** and **executes programs** on the Solana blockchain.

- It encompasses everything from **transaction validation** to **program execution**, acting as the bridge between the network layer and the on-chain state.

- When executing sBPF programs (solana smart contracts), the **BPF Loader** program handles the complexity of running untrusted code.

- It serializes the **instruction's accounts and data** into a linear memory buffer known as the **Program Input**.

- The Solana Runtime is the core **execution environment** that processes **transactions** and **executes programs** on the Solana blockchain.

- It encompasses everything from **transaction validation** to **program execution**, acting as the bridge between the network layer and the on-chain state.

- When executing sBPF programs (solana smart contracts), the **BPF Loader** program handles the complexity of running untrusted code.

- It serializes the **instruction's accounts and data** into a linear memory buffer known as the **Program Input**.

- The **Program Input** is then passed to the **SVM**, which executes **sBPF bytecode** while enforcing compute unit limits and validating memory accesses

Transactions

- Solana's execution model is built on **transactions** containing **instructions**.

- A Solana **instruction** is the **smallest** unit of **execution** logic on the Solana blockchain. It's essentially a command that tells a program what operation to perform

- The EVM has a **single-threaded** execution environment, meaning it processes transactions in a strictly sequential manner: limiting itself to a single CPU core of the validator.

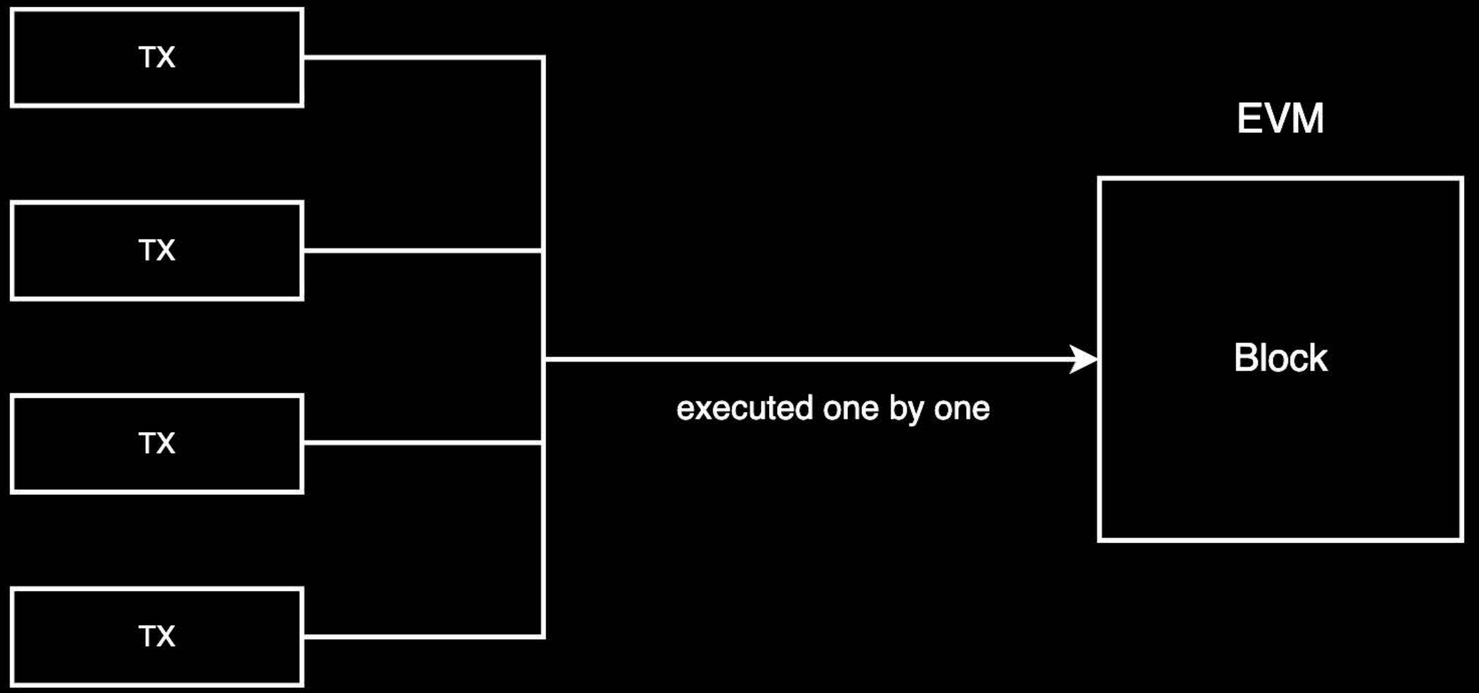
- The EVM has a **single-threaded** execution environment, meaning it processes transactions in a strictly sequential manner: limiting itself to a single CPU core of the validator.

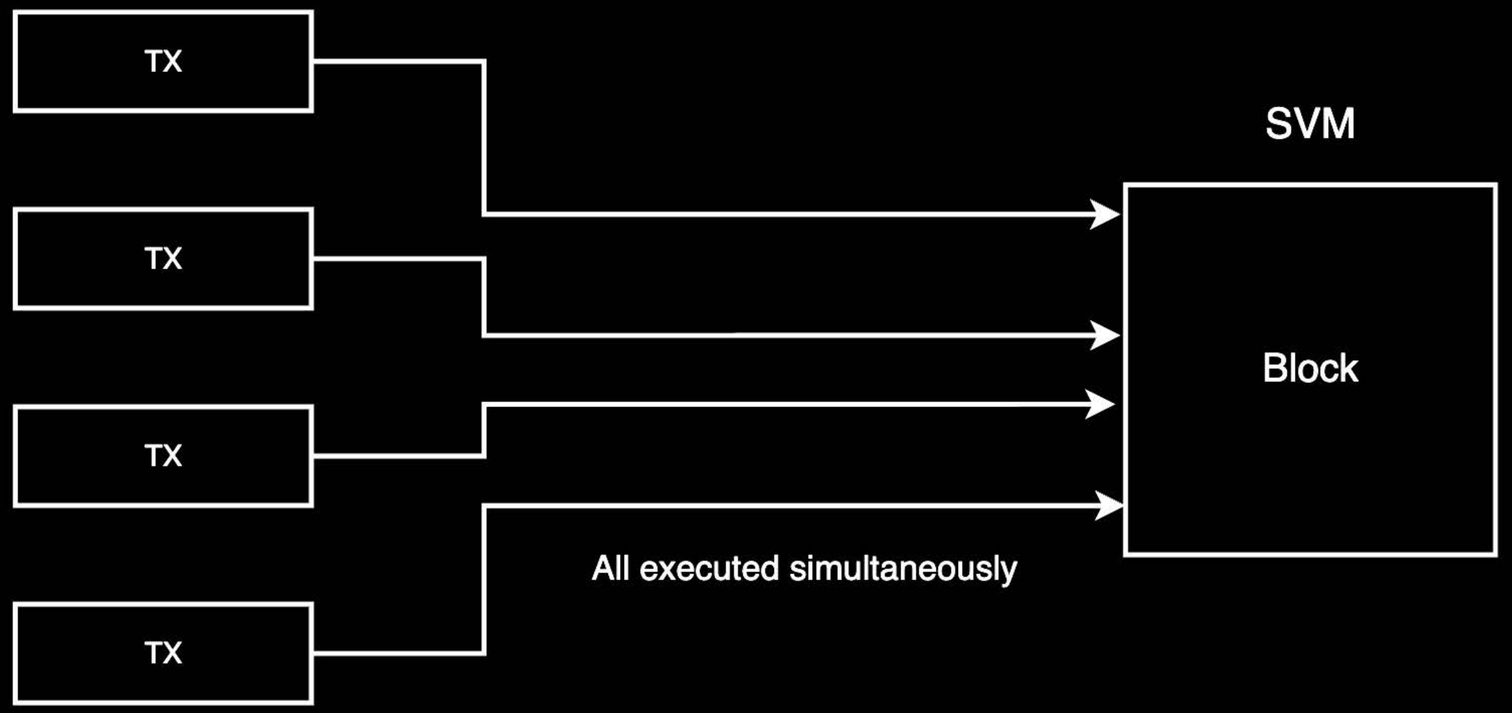
- In this model, interactions with the blockchain state such as reading or modifying balances in a contract are implicit, meaning the contract stores and accesses data during execution without previously declaring which resources will be affected.

- Solana adopts a **multi-threaded** approach, where each transaction must be explicitly declared before any execution, knowing which data will be read and which will be written or modified.

- Solana adopts a **multi-threaded** approach, where each transaction must be explicitly declared before any execution knowing which data will be read and which will be written or modified.

- This is possible thanks to Sealevel, the integrated parallel processing engine, which enables the SVM to handle tens of thousands of transactions simultaneously.





Solana's Account Model

- Accounts are the fundamental data storage unit in Solana.
- Every piece of data lives in an account.
- A program can **only** modify accounts it owns

- In Solana, all state is stored in accounts with 32-byte addresses.
- Accounts can store up to 10 MiB of data.
- They require a rent deposit in lamports (SOL) that is proportional to the amount of data stored.
- Each account has a program owner.

Account structure

Account structure

- **Lamports:**
  The smallest unit of SOL (1 SOL = 1 billion lamports)

- **Owner:**
  The ID of the program that owns and controls this account.

- **Executable**:
  This field indicates whether an account is an executable program.

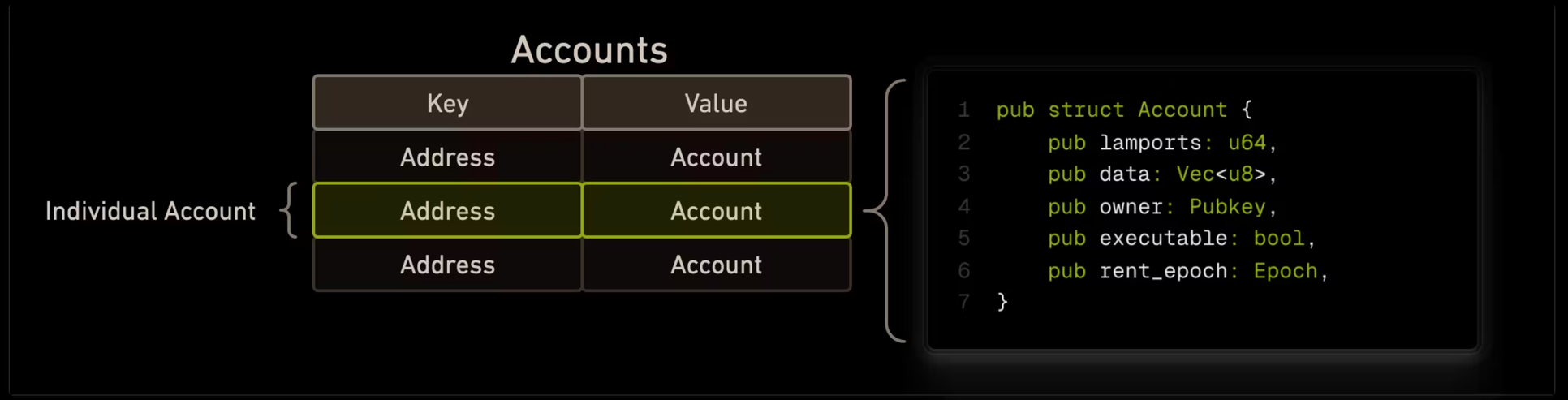
- **Data:**
  An array of bytes that stores arbitrary data for an account:
    For executable / non-executable accounts

- **Rent Epoch:**
  A legacy field that is no longer used, it tracked when an account would need to pay rent to keep its data on the network.

Public Key:
"14grJpemFaf88c8tiVb77W7TYg2W3ir6pfkKz3YjhhZ5"

Address

Account



https://solana.com/docs/core/accounts

INVERSIVE LABS

In Solana, program code and state are stored in separate accounts for efficiency

- **BPF Loader Account**: Owns executable code
  - *executable*: true

- **Program Account**:
  - Account's *data* holds the program bytecode.
  - Owned by BPF Loader

- **Data Account**: Stores state (e.g., balances)
  - *executable*: false
  - Owned by the Program.

In Solana, program code and state are stored in separate accounts for efficiency

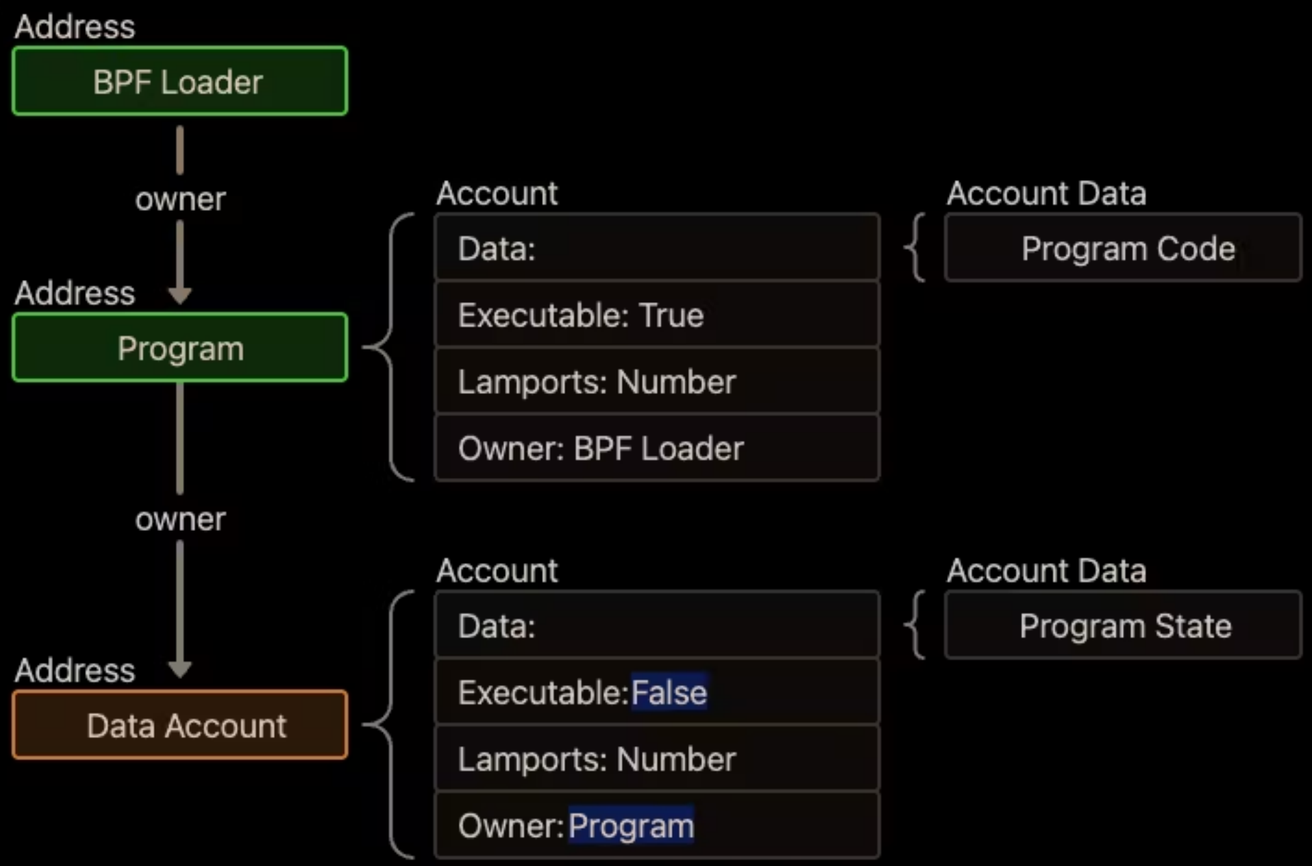
- **BPF Loader Account**: Owns executable code
  - *executable*: true

- **Program Account**:
  - Account's *data* holds the program bytecode.
  - Owned by BPF Loader

- **Data Account**: Stores state (e.g., balances)
  - *executable*: false
  - Owned by the Program.

Steps for creating an account:

1. System Program creates account.
2. Then transfers ownership and initializes data.



https://solana.com/docs/core/accounts#data-account

Account Types

- **System Program:**
  - Creation of new accounts
  - Space allocation
  - Program ownership assignment
  - SOL transfers

- **System Program:**
  - Creation of new accounts
  - Space allocation
  - Program ownership assignment
  - SOL transfers

- **Sysvar Accounts:**
  - These are accounts located at predefined addresses that provide access to cluster state data.

- **System Program:**
  - Creation of new accounts
  - Space allocation
  - Program ownership assignment
  - SOL transfers

- **Sysvar Accounts:**
  - These are accounts located at predefined addresses that provide access to cluster state data.

- **Program Account:**
  - Commonly called "Program ID", stores the program's executable code.

- **System Program:**
  - Creation of new accounts
  - Space allocation
  - Program ownership assignment
  - SOL transfers

- **Sysvar Accounts:**
  - These are accounts located at predefined addresses that provide access to cluster state data.

- **Program Account:**
  - Commonly called "Program ID", stores the program's executable code.

- **Data Accounts:**
  - A program's executable code is stored in a different account from the program's state.

- **System Program:**
  - Creation of new accounts
  - Space allocation
  - Program ownership assignment
  - SOL transfers

- **Sysvar Accounts:**
  - These are accounts located at predefined addresses that provide access to cluster state data.

- **Program Account:**
  - Commonly called "Program ID", stores the program's executable code.

- **Data Accounts:**
  - A program's executable code is stored in a different account from the program's state.

- **PDA:**
  - PDAs are deterministically derived addresses using a combination of predefined seeds, a bump seed, and a program ID.

Instructions

- When a user submits a **transaction**, it contains one or more instructions that execute **sequentially**.

- Each instruction invokes a specific program's **entrypoint** with the provided accounts and data

The structure of an instruction consists of:

- **Program Address**:
  - The address of the program containing the logic for the instruction being invoked

The structure of an instruction consists of:

- **Program Address**:
  - The address of the program containing the logic for the instruction being invoked

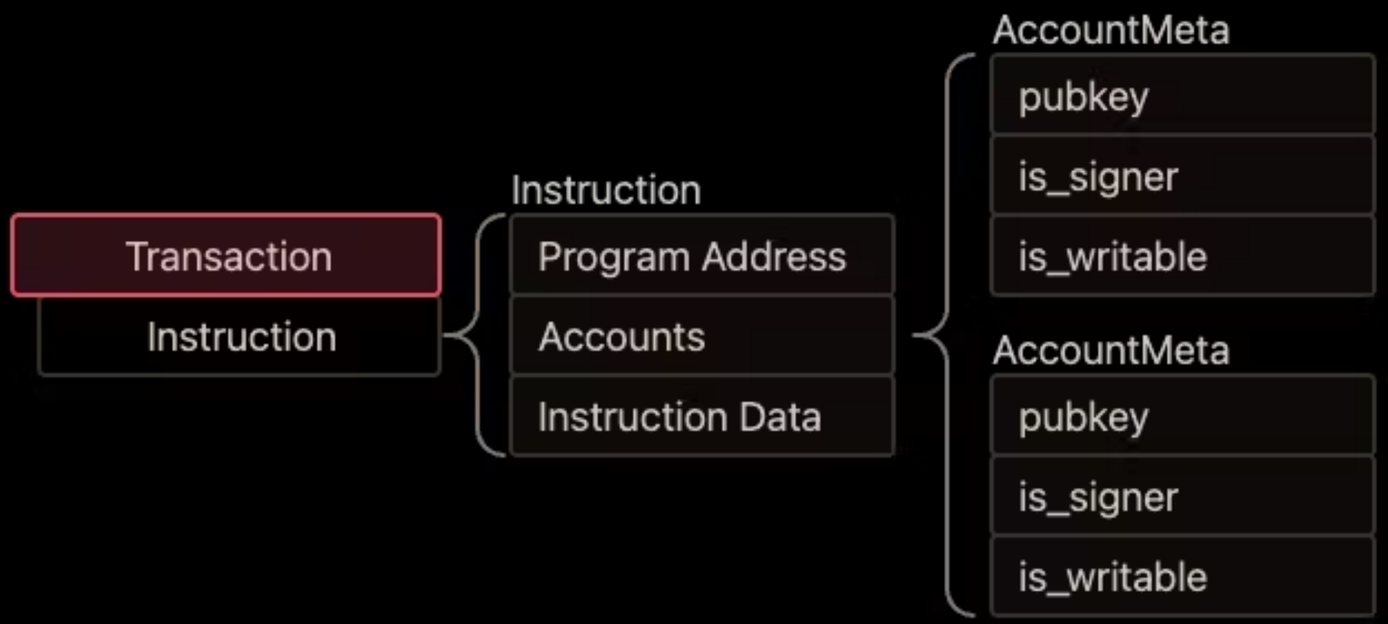
- **Account Metadata**:
  - A list of all the accounts that the instructions reads from or writes to. Each account metadata instance consists of:
    - *pubkey*
    - *is_signer*
    - *is_writable*

The structure of an instruction consists of:

- **Program Address**:
  - The address of the program containing the logic for the instruction being invoked

- **Account Metadata**:
  - A list of all the accounts that the instructions reads from or writes to. Each account metadata instance consists of:
    - *pubkey*
    - *is_signer*
    - *is_writable*

- **Instruction Data**:
  - An array of bytes that specifies which instruction to invoke in the program



https://solana.com/docs/core/transactions#accountmeta

INVERSIVE LABS

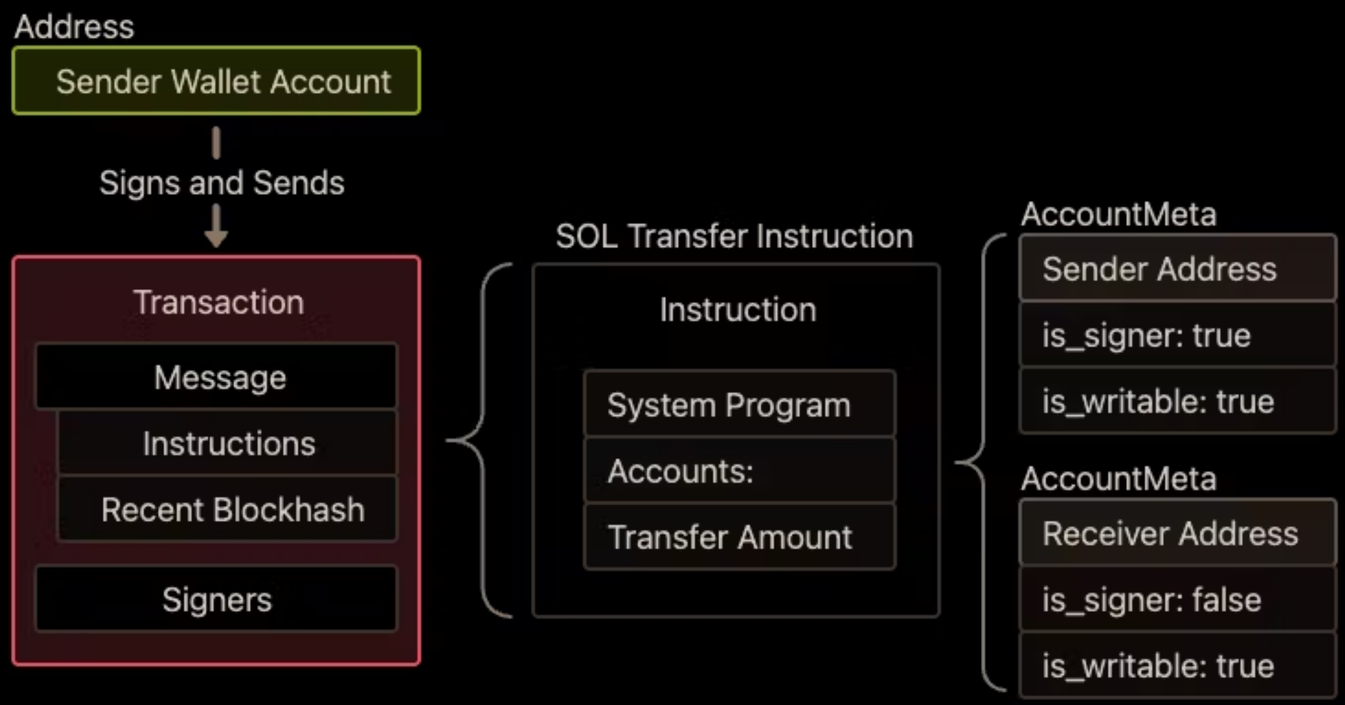
- In Solana, wallets are accounts that belong to the System Program.
- Only the owner can change the account data, and transferring SOL requires a transaction to invoke the System Program.

- In Solana, wallets are accounts that belong to the System Program.
- Only the owner can change the account data, and transferring SOL requires a transaction to invoke the System Program.

For a successful transaction to take place, the following must hold:

- Sender's account:
  - *is_signer* = true:
    - Allows the System Program to deduct its lamports balance.
  - *is_writable* = true:
    - Allows the account to be modified.

- Recipient's account:
  - *is_signer* = false
  - *is_writeable* = true:
    - Allows the account to be modified.



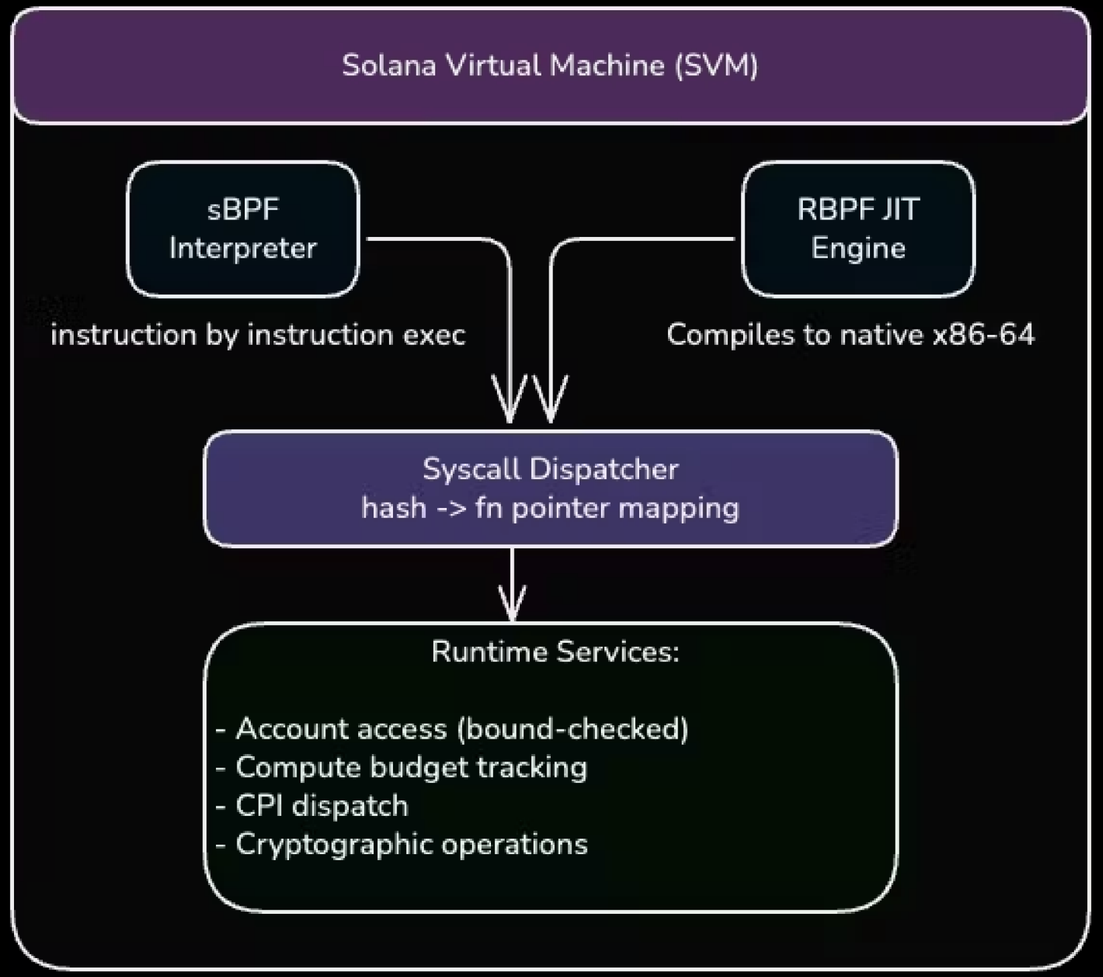
https://solana.com/docs/core/transactions#sol-transfer-example

The Solana Virtual Machine

- Deployed by the BPF Loader program to execute a client program
- Runs sBPF bytecode in a sandboxed environment
- Programs cannot directly access OS or other programs
- All external interactions go through syscalls

Core Components:

- **sBPF Interpreter**: Instruction-by-instruction execution
- **RBPF JIT Engine**: Compiles to native x86-64 code (Agave)
- **Syscall Dispatcher**: Hash → function pointer mapping
- **Runtime Services**:
  - Bounds-checked account access
  - Compute budget tracking
  - CPI dispatch
  - Cryptographic operations



Memory Model

- Solana's Memory Model in the SVM partitions a 64-bit virtual address space into fixed regions for secure, parallel execution

- Bytecode runs via LLVM-generated sBPF, JIT-compiled in validators (rBPF in Agave), with syscalls bridging VM-host.
  - BlueShift has developed an alternative approach that allows developers to build sBPF programs using standard Rust tooling, eliminating the need for Solana's forked Rust and LLVM.

0x100000000 (Program Memory, R-X):

- Loads ELF binary
  - .text for code
  - .rodata for constants/strings)

0x200000000 (Stack, RW, 4KB fixed):

- Register r10 (frame pointer)
- Grows downward (toward lower addresses)

0x300000000 (Heap, RW, 32KB default/256KB max):

- Dynamic memory
- Used to be managed via sol_alloc_free_ syscall but is now deprecated

0x400000000 (Input Region, RW):

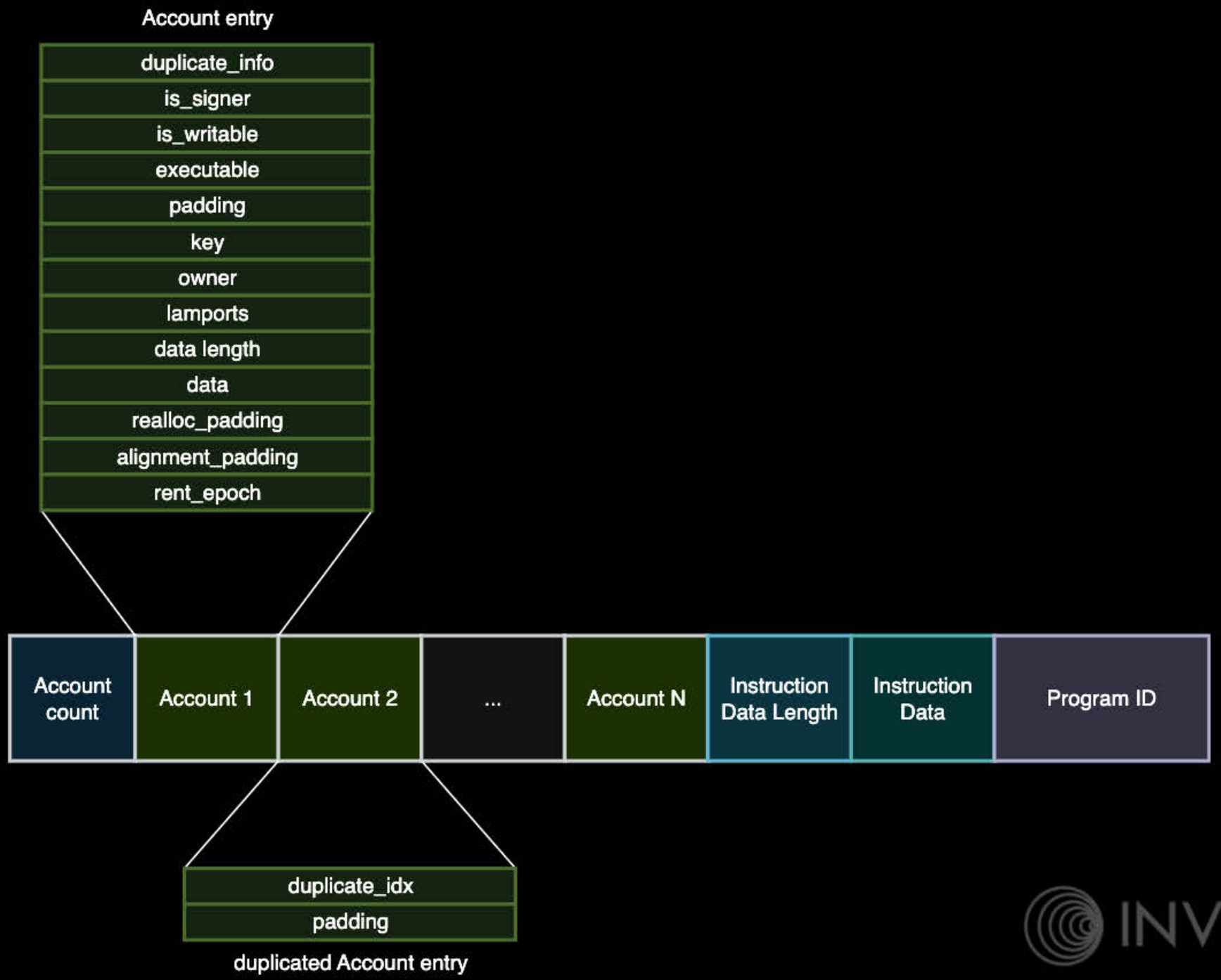
- Region for transaction input data
- Also known as the Program Input
- This region contains serialized transaction data, such as account and instruction data

Serialization

- When a program is invoked, the entry point receives a single pointer to a serialized buffer.
- This serialized buffer is commonly known as the **Program Input**
- The serialization format it's Solana-specific

Program Input Layout:

- **Account Count**: 8 bytes.
- **Account Array:**
  - Accounts can be either unique or duplicated entries in the Program Input
    - this is denoted by the *duplicate_info* field
  - **10KB Padding**: Reserves 10,240 bytes per account for growth up to 10KB without reallocation.
- **Instruction Data**: 8 bytes length + N bytes of data.
- **Program ID**: 32 bytes.


Account entry
duplicate_info
is_signer
is_writable
executable
padding
key
owner
lamports
data length
data
realloc_padding
alignment_padding
rent_epoch
Account count
Account 1
Account 2
...
Account N
Instruction Data Length
Instruction Data
Program ID
duplicate_idx
padding
duplicated Account entry

Cross Program Invocation

- A Cross Program Invocation (CPI) refers to when one program invokes the instructions of another program, enabling program composition in Solana.

- Internally, each **CPI** instruction must specify:
  - **Program Address**: Identifies the program to invoke

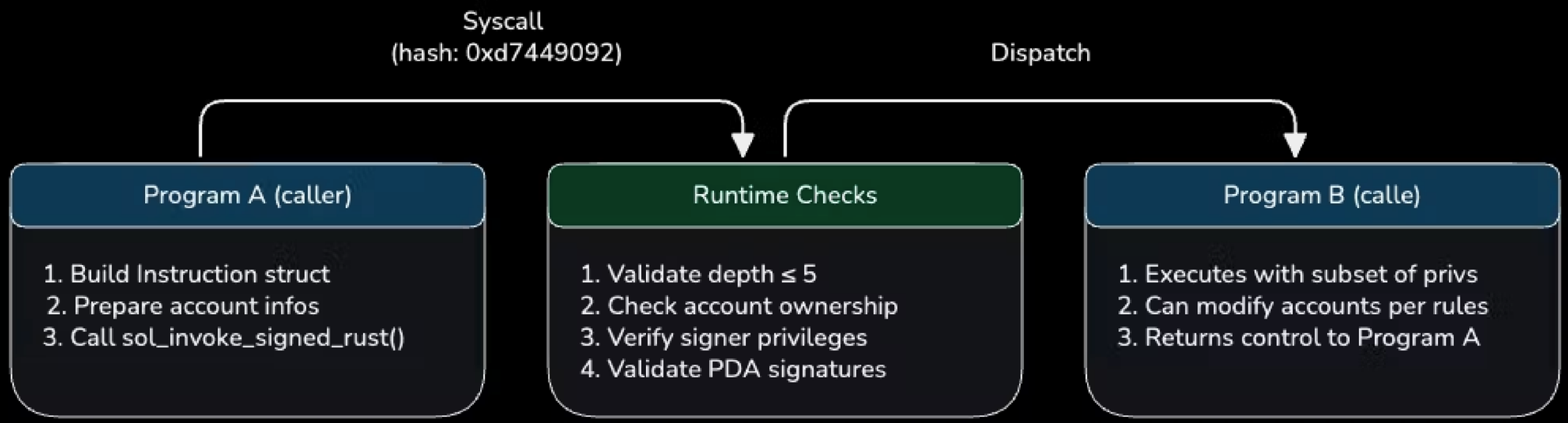
  - **Accounts**: Lists every account the instruction reads from or writes to, including other programs

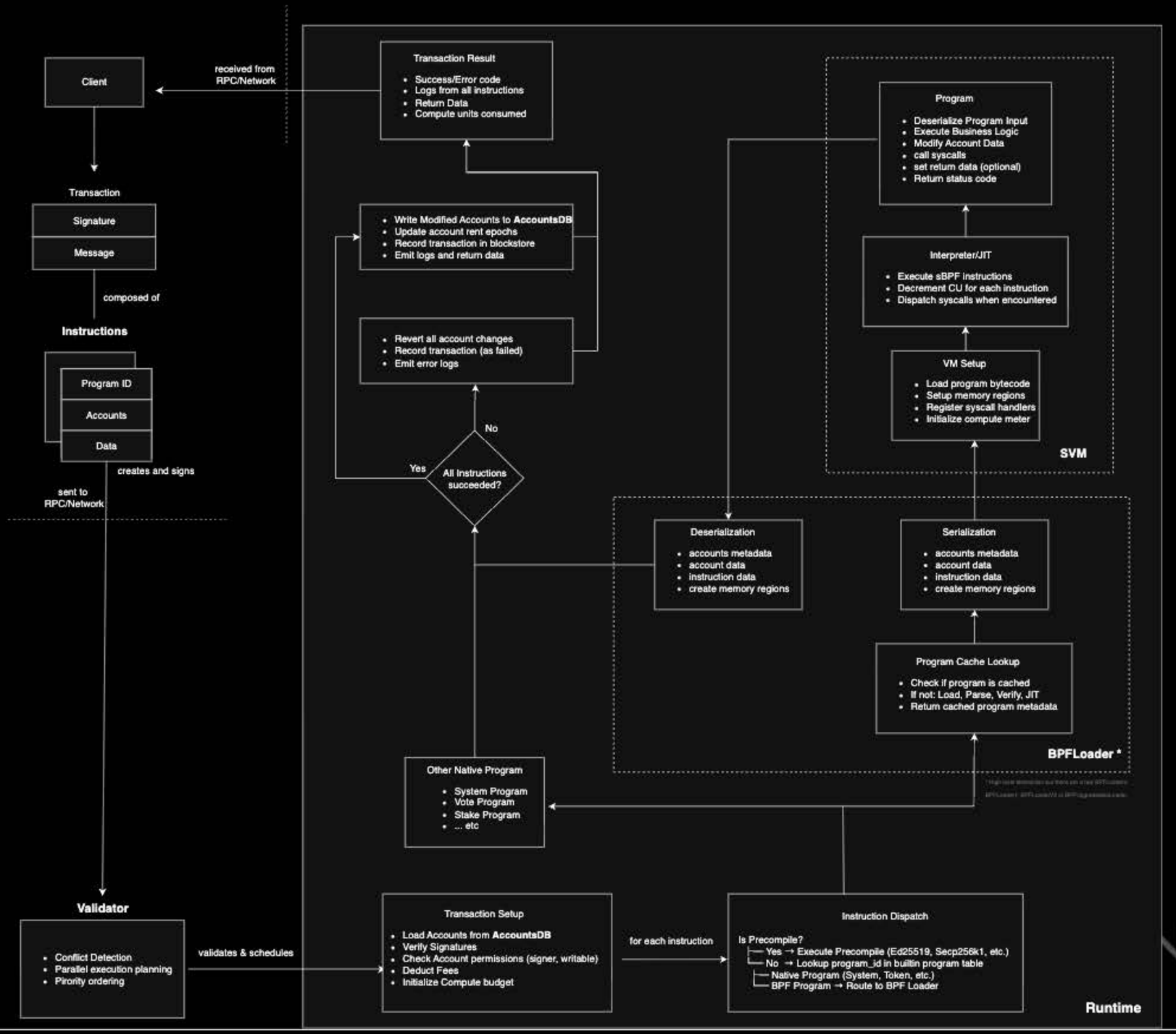
  - **Instruction Data**: Specifies which instruction to invoke in the program, along with any data the instruction requires (function arguments)

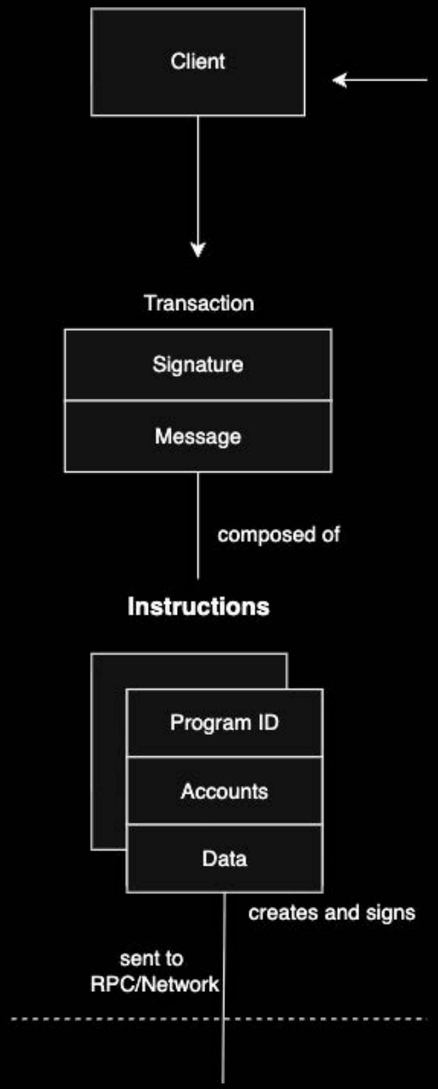
- When a program performs a **CPI** to another program:
  - The privileges of the initial transaction signer extend to the invoked program (e.g., A->B)

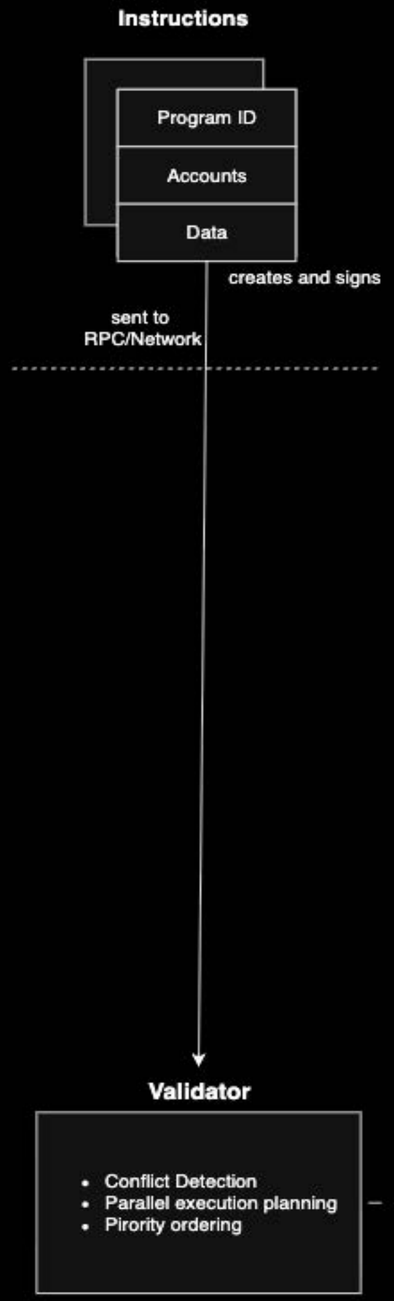
  - The invoked program can perform additional CPIs to other programs, up to a depth of 4
    - Program A → Program B → Program C → Program D → Program E

  - Programs can "sign" on behalf of PDAs derived from their program ID

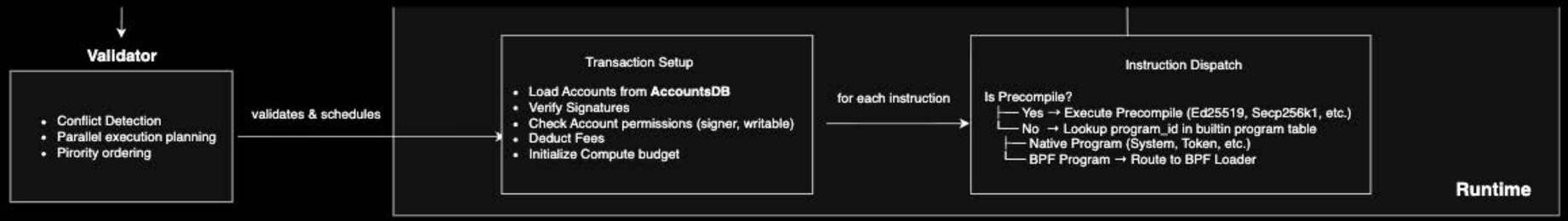


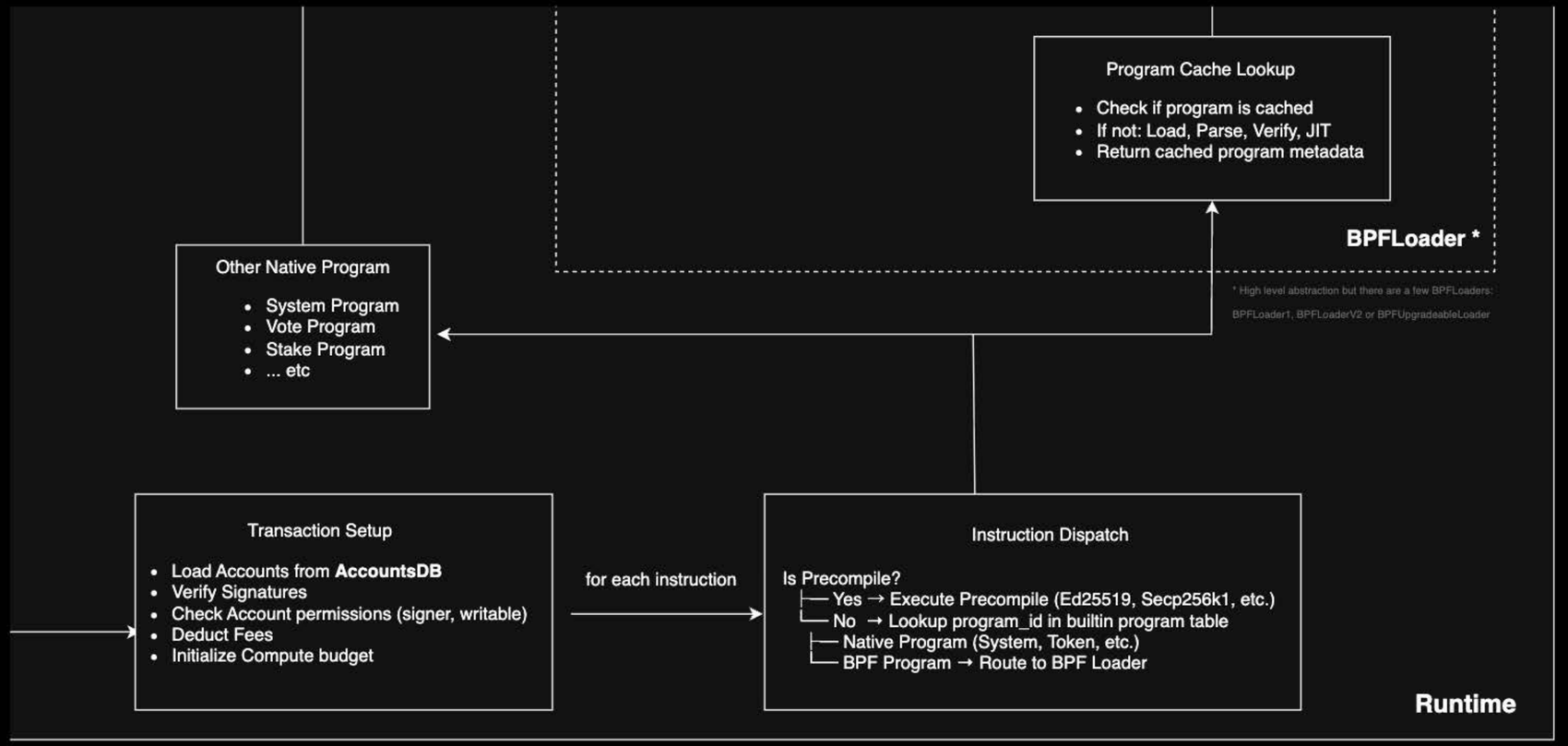


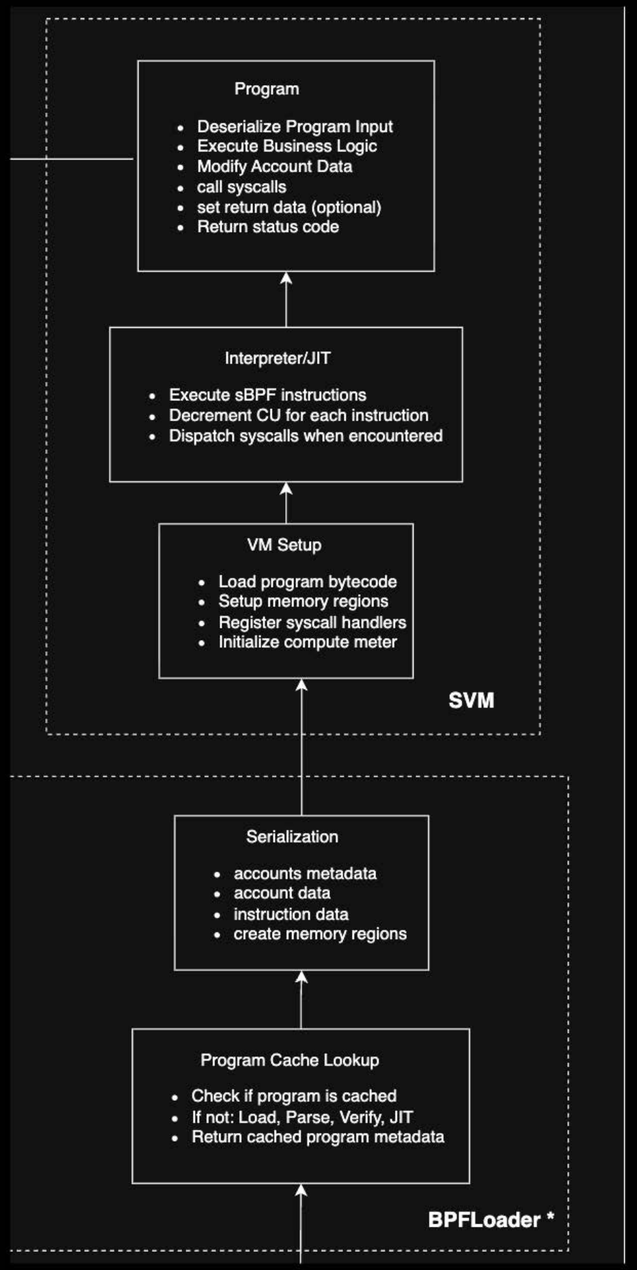


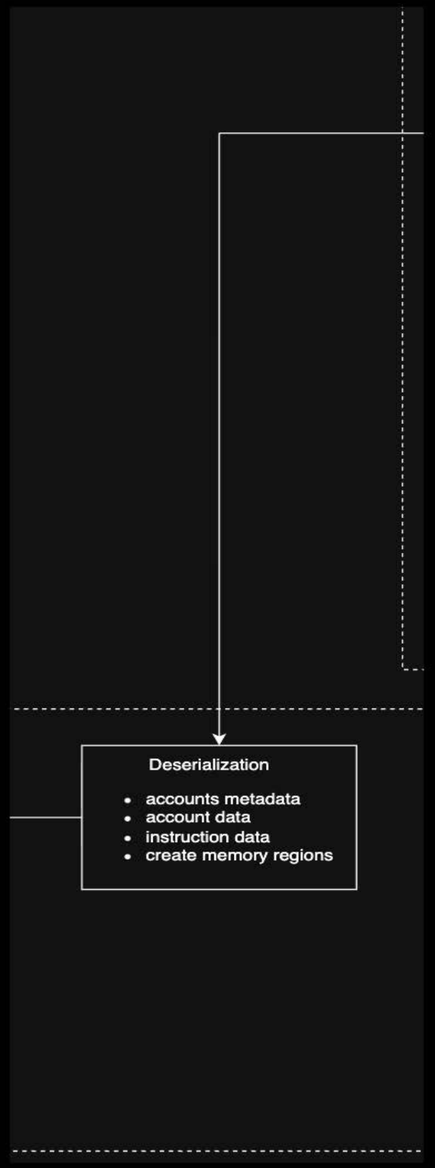


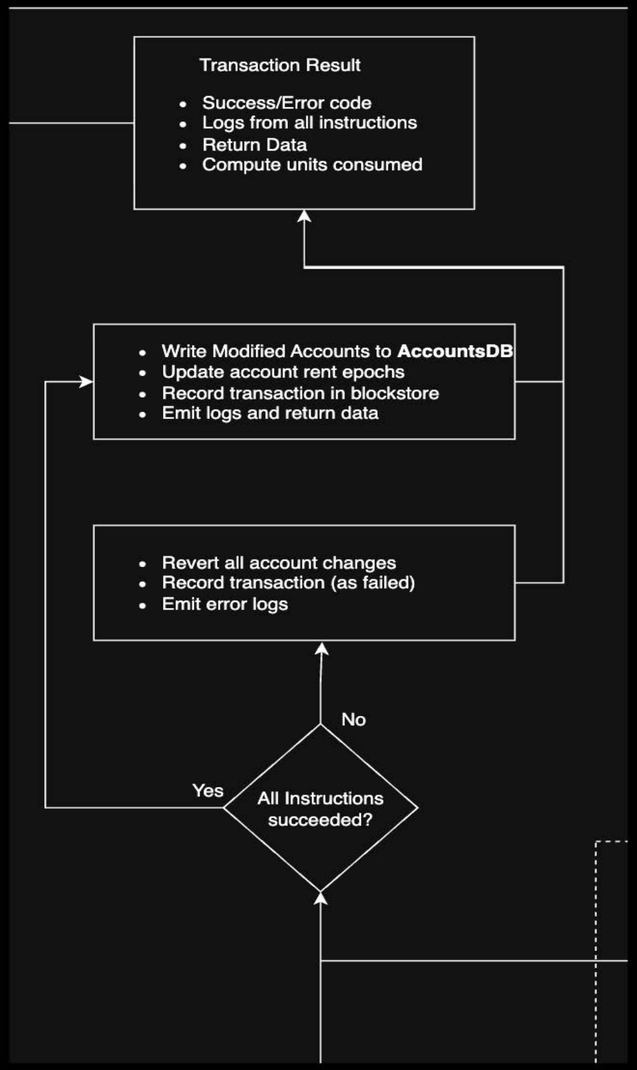
INVERSIVE LABS











INVERSIVE LABS

3. The sBPF Architecture

- sBPF (Solana BPF) is a **modified, high-performance version of eBPF** (originally from the Linux kernel) designed to run **untrusted programs** safely inside Solana's validator nodes.

- Solana Programs are compiled as **ELF binaries** containing **sBPF bytecode**.

Aspect	eBPF (Linux)	sBPF (Solana)
Syscall dispatch	Integer IDs (bpf_trace_printk = 6)	32-bit murmur3 hash (sol_log_ = 0x207559bd)
Memory model	Kernel-specific maps	Fixed 64-bit virtual address space
ELF format	Standard relocations	Custom R_BPF_* relocations
Execution	Kernel context	Fully sandboxed VM
Versioning	Single version	Multiple incompatible versions (v0–v4)

- The sBPF ISA is based on **eBPF with Solana-specific extensions**, using a fixed 64-bit instruction encoding (except lddw instruction on v0-v1) with multiple instruction classes
- Each sBPF instruction is a **64-bit word** (sBPF v2+) with the following bit layout

Bits	Field	Description
7:0	opcode	8-bit opcode determining instruction class and operation
11:8	dst_reg	4-bit destination register (0-10)
15:12	src_reg	4-bit source register (0-10)
31:16	offset	16-bit signed offset for memory/branch operations
63:32	imm	32-bit immediate value

Opcode Structure

The opcode byte is further subdivided based on instruction class :

For Normal Instructions (ALU/JMP)

- **Bits 2:0**: Instruction class (ALU/JMP/JMP32/ALU64)
- **Bit 3**: Source mode (0=immediate, 1=register)
- **Bits 7:4**: Operation mode (ADD/SUB/MUL/DIV/OR/AND/LSH/RSH/etc.)

For Memory Instructions (LD/LDX/ST/STX)

- **Bits 2:0**: Instruction class
- **Bits 4:3**: Size mode (0=word/32-bit, 1=half/16-bit, 2=byte/8-bit, 3=double/64-bit)
- **Bits 7:5**: Addressing mode (IMM/MEM/etc.)

Instruction Classes

sBPF defines **8 instruction classes** based on bits [2:0] of the opcode:

Class	Value	Purpose
LD	0x0	Load immediate (64-bit LDDW)
LDX	0x1	Load from memory
ST	0x2	Store immediate to memory
STX	0x3	Store register to memory
ALU	0x4	32-bit ALU operations
JMP	0x5	Conditional/unconditional jumps
JMP32	0x6	32-bit conditional jumps
ALU64	0x7	64-bit ALU operations

Registers

The sBPF instruction set defines 11 general purpose registers, each 64 bits wide.

These 11 GPR have specific conventional uses:

- **r0**: Return value register - holds the result of function calls and syscalls
- **r1-r5**: Argument registers - used to pass parameters to functions and syscalls
  - At solana program entrypoint, r1 holds a pointer to the Program Input
- **r6-r9**: Callee-saved registers - preserved across function calls via the shadow stack
- **r10**: Frame pointer - points to the current stack frame

sBPF Versions

- There is a total of 4 different sBPF versions.

- Solana evolved the sBPF ISA with extensions to fix security issues and align with formal verification efforts.

- **SIMD** = Solana Improvement Documents
  - Governance Proposals for protocol changes
  - Analogous to EIP in Ethereum (Ethereum Improvement Proposals)

- **V0** - Legacy version (almost eBPF compatible)
- **V1** - Dynamic Stack Frames (*SIMD-0166*)
- **V2** - Major Instruction Changes
  - Replacement of 16-byte lddw instruction (*SIMD-0173)*
  - Better wide multiplication, signed division and explicit sign extension (*SIMD-0174)*
- **V3** - Programs are linked at compile time
  - Relocations are no longer needed
  - Syscalls are now static (*SIMD-0178)*
  - Stronger ELF validation *(SIMD-0179)*
- **V4** - Currently on development

https://github.com/solana-foundation/solana-improvement-documents/

V0

```
0x1000007388    bf12000000000000    mov64 r2, r1
0x1000007390    bfa1000000000000    mov64 r1, r10
0x1000007398    07010000b8ffffff    add64 r1, 0xffffffb8
0x10000073a0    8510000b6200000     call fcn.1000017958
0x10000073a8    79a6c8ff00000000    ldxdw r6, [r10+0xffc8]
0x10000073b0    79a7c0ff00000000    ldxdw r7, [r10+0xffc0]
0x10000073b8    79a2d0ff00000000    ldxdw r2, [r10+0xffd0]
0x10000073c0    79a1d8ff00000000    ldxdw r1, [r10+0xffd8]
0x10000073c8    79a3e0ff00000000    ldxdw r3, [r10+0xffe0]
0x10000073d0    7b3a08f000000000    stxdw [r10+0xf008], r3
0x10000073d8    7b1a00f000000000    stxdw [r10+0xf000], r1
```

V1

```
0x1000007408    070a000080ffffff    add64 r10, 0xffffff80
0x1000007410    bf12000000000000    mov64 r2, r1
0x1000007418    bfa1000000000000    mov64 r1, r10
0x1000007420    0701000038000000    add64 r1, 0x38
0x1000007428    85100000fc200000    call 0x1000017c10
0x1000007430    79a5580000000000    ldxdw r5, [r10+0x58]
0x1000007438    79a6480000000000    ldxdw r6, [r10+0x48]
0x1000007440    79a7400000000000    ldxdw r7, [r10+0x40]
0x1000007448    79a2500000000000    ldxdw r2, [r10+0x50]
0x1000007450    79a1600000000000    ldxdw r1, [r10+0x60]
0x1000007458    7b1af8ff00000000    stxdw [r10+0xfff8], r1
```

INVERSIVE LABS

V0

```
0x1000007388    bf12000000000000    mov64 r2, r1
0x1000007390    bfa1000000000000    mov64 r1, r10
0x1000007398    07010000b8ffffff    add64 r1, 0xffffffb8
0x10000073a0    8510000000b62000    call fcn.1000017958
0x10000073a8    79a6c8ff00000000    ldxdw r6, [r10+0xffc8]
0x10000073b0    79a7c0ff00000000    ldxdw r7, [r10+0xffc0]
0x10000073b8    79a2d0ff00000000    ldxdw r2, [r10+0xffd0]
0x10000073c0    79a1d8ff00000000    ldxdw r1, [r10+0xffd8]
0x10000073c8    79a3e0ff00000000    ldxdw r3, [r10+0xffe0]
0x10000073d0    7b3a08f000000000    stxdw [r10+0xf008], r3
0x10000073d8    7b1a00f000000000    stxdw [r10+0xf000], r1
```

V1

```
0x1000007408    070a000080ffffff    add64 r10, 0xffffff80
0x1000007410    bf12000000000000    mov64 r2, r1
0x1000007418    bfa1000000000000    mov64 r1, r10
0x1000007420    0701000038000000    add64 r1, 0x38
0x1000007428    8510000000fc2000    call 0x1000017c10
0x1000007430    79a5580000000000    ldxdw r5, [r10+0x58]
0x1000007438    79a6480000000000    ldxdw r6, [r10+0x48]
0x1000007440    79a7400000000000    ldxdw r7, [r10+0x40]
0x1000007448    79a2500000000000    ldxdw r2, [r10+0x50]
0x1000007450    79a1600000000000    ldxdw r1, [r10+0x60]
0x1000007458    7b1af8ff00000000    stxdw [r10+0xfff8], r1
```

INVERSIVE LABS

V1

```
0x1000007408    070a000080ffffff   add64 r10, 0xffffff80
0x1000007410    bf12000000000000   mov64 r2, r1
0x1000007418    bfa1000000000000   mov64 r1, r10
0x1000007420    0701000038000000   add64 r1, 0x38
0x1000007428    8510000fc2000000   call 0x1000017c10
0x1000007430    79a5580000000000   ldxdw r5, [r10+0x58]
0x1000007438    79a6480000000000   ldxdw r6, [r10+0x48]
0x1000007440    79a7400000000000   ldxdw r7, [r10+0x40]
0x1000007448    79a2500000000000   ldxdw r2, [r10+0x50]
0x1000007450    79a1600000000000   ldxdw r1, [r10+0x60]
0x1000007458    7b1af8ff00000000   stxdw [r10+0xfff8], r1
0x1000007460    bfa1000000000000   mov64 r1, r10
0x1000007468    0701000038000000   add64 r1, 0x38
0x1000007470    bf73000000000000   mov64 r3, r7
0x1000007478    bf64000000000000   mov64 r4, r6
0x1000007480    85100000d3f7ffff   call 0x1000003320
0x1000007488    b708000000000000   mov64 r8, 0x0
0x1000007490    180100001a0000..   lddw r1, 0x800000000000001a
```

V2

```
0x1000007410    070a000080ffffff   add64 r10, 0xffffff80
0x1000007418    bf12000000000000   mov64 r2, r1
0x1000007420    bfa1000000000000   mov64 r1, r10
0x1000007428    0701000038000000   add64 r1, 0x38
0x1000007430    851000002b210000   call 0x1000017d90
0x1000007438    9ca5580000000000   ldxq r5, [r10+0x58]
0x1000007440    9ca6480000000000   ldxq r6, [r10+0x48]
0x1000007448    9ca7400000000000   ldxq r7, [r10+0x40]
0x1000007450    9ca2500000000000   ldxq r2, [r10+0x50]
0x1000007458    9ca1600000000000   ldxq r1, [r10+0x60]
0x1000007460    9f1af8ff00000000   stxq [r10-0x8], r1
0x1000007468    bfa1000000000000   mov64 r1, r10
0x1000007470    0701000038000000   add64 r1, 0x38
0x1000007478    bf73000000000000   mov64 r3, r7
0x1000007480    bf64000000000000   mov64 r4, r6
0x1000007488    85100000b7f7ffff   call 0x1000003248
0x1000007490    b708000000000000   mov64 r8, 0x0
0x1000007498    b4010000 1a000000  mov r1, 0x1a
0x10000074a0    f701000000000080   hor64 r1, 0x80000000
```

INVERSIVE LABS

V1

```
0x1000007408    070a000080ffffff   add64 r10, 0xffffff80
0x1000007410    bf12000000000000   mov64 r2, r1
0x1000007418    bfa1000000000000   mov64 r1, r10
0x1000007420    0701000038000000   add64 r1, 0x38
0x1000007428    85100000fc200000   call 0x1000017c10
0x1000007430    79a5580000000000   ldxdw r5, [r10+0x58]
0x1000007438    79a6480000000000   ldxdw r6, [r10+0x48]
0x1000007440    79a7400000000000   ldxdw r7, [r10+0x40]
0x1000007448    79a2500000000000   ldxdw r2, [r10+0x50]
0x1000007450    79a1600000000000   ldxdw r1, [r10+0x60]
0x1000007458    7b1af8ff00000000   stxdw [r10+0xfff8], r1
0x1000007460    bfa1000000000000   mov64 r1, r10
0x1000007468    0701000038000000   add64 r1, 0x38
0x1000007470    bf73000000000000   mov64 r3, r7
0x1000007478    bf64000000000000   mov64 r4, r6
0x1000007480    85100000d3f7ffff   call 0x1000003320
0x1000007488    b708000000000000   mov64 r8, 0x0
0x1000007490    180100001a0000..   lddw r1, 0x800000000000001a
```

V2

```
0x1000007410    070a000080ffffff   add64 r10, 0xffffff80
0x1000007418    bf12000000000000   mov64 r2, r1
0x1000007420    bfa1000000000000   mov64 r1, r10
0x1000007428    0701000038000000   add64 r1, 0x38
0x1000007430    851000002b210000   call 0x1000017d90
0x1000007438    9ca5580000000000   ldxq r5, [r10+0x58]
0x1000007440    9ca6480000000000   ldxq r6, [r10+0x48]
0x1000007448    9ca7400000000000   ldxq r7, [r10+0x40]
0x1000007450    9ca2500000000000   ldxq r2, [r10+0x50]
0x1000007458    9ca1600000000000   ldxq r1, [r10+0x60]
0x1000007460    9f1af8ff00000000   stxq [r10-0x8], r1
0x1000007468    bfa1000000000000   mov64 r1, r10
0x1000007470    0701000038000000   add64 r1, 0x38
0x1000007478    bf73000000000000   mov64 r3, r7
0x1000007480    bf64000000000000   mov64 r4, r6
0x1000007488    85100000b7f7ffff   call 0x1000003248
0x1000007490    b708000000000000   mov64 r8, 0x0
0x1000007498    b4010000 1a000000  mov r1, 0x1a
0x10000074a0    f701000000000080   hor64 r1, 0x80000000
```

INVERSIVE LABS

V1

```
7901ffff00000000    ldxdw r1, [r0+0xffff]
8d0000000200000     callx r2
bfa1000000000000    mov64 r1, r10
```

V2

```
9c01ffff00000000    ldxq r1, [r0-0x1]
8d2000000000000     callx r2
bfa1000000000000    mov64 r1, r10
```

V1

```
7901ffff00000000    ldxdw r1, [r0+0xffff]
8d0000000200000     callx r2
bfa1000000000000    mov64 r1, r10
```

V2

```
9c01ffff00000000    ldxq r1, [r0-0x1]
8d2000000000000     callx r2
bfa1000000000000    mov64 r1, r10
```

V0-V2

```
79a3200000000000    ldxdw r3, [r10+0x20]
7b2308000000000     stxdw [r3+0x8], r2
7b1300000000000     stxdw [r3], r1
070a000000010000    add64 r10, 0x100
9500000000000000    exit
```

V3

```
9ca3200000000000    ldxq r3, [r10+0x20]
9f2308000000000     stxq [r3+0x8], r2
9f1300000000000     stxq [r3+0x0], r1
070a000000010000    add64 r10, 0x100
9d0000000000000     exit
```

V0-V2

```
79a3200000000000    ldxdw r3, [r10+0x20]
7b2308000000000     stxdw [r3+0x8], r2
7b1300000000000     stxdw [r3], r1
070a000000010000    add64 r10, 0x100
9500000000000000    exit
```

V3

```
9ca3200000000000    ldxq r3, [r10+0x20]
9f2308000000000     stxq [r3+0x8], r2
9f1300000000000     stxq [r3+0x0], r1
070a000000010000    add64 r10, 0x100
9d0000000000000     exit
```

sBPF Relocations

- sBPF relocations are a **limited, non-standard dynamic relocation mechanism** used to fix up Clang-generated shared objects for execution in the Solana BPF VM

- The key difference from standard ELF relocations is that sBPF relocations **violate both eBPF and ELF specs** and are specifically designed to shift program code from zero-based addressing to the **MM_PROGRAM** segment at virtual address **0x100000000**, aka the image base of the program for sBPF v0, v1 and v2

- This changes for **v3** are reverted back to zero-based addressing, as **MM_PROGRAM** becomes **0**, and there are **no** relocations

- sBPF supports exactly **three** relocation types, each with Solana-specific behavior
    - **R_BPF_64_64**
    - **R_BPF_64_RELATIVE**
    - **R_BPF_64_32**

1. **R_BPF_64_64**

Sets an absolute address of a symbol as the 64-bit immediate field of an lddw instruction. The relocation:

- Reads the implicit addend from the immediate fields at offsets r_offset+4 and r_offset+12
- Looks up the symbol value S from the dynamic symbol table
- Computes V = S + A, adding MM_PROGRAM_ADDR if the result is below it
- Writes back the low and high 32-bit parts

```
b4020000a8d90200    mov r2, 0x2d9a8          ; RELOC 64
f702000000000000    hor64 r2, 0x0
bfa1000000000000    mov64 r1, r10
0701000068010000    add64 r1, 0x168
b703000001000000    mov64 r3, 0x1
bf74000000000000    mov64 r4, r7
```

```
b4020000a8d90200    mov r2, 0x2d9a8          ; RELOC 64
f702000000000000    hor64 r2, 0x0
bfa1000000000000    mov64 r1, r10
0701000068010000    add64 r1, 0x168
b703000001000000    mov64 r3, 0x1
bf74000000000000    mov64 r4, r7
```

2. **R_BPF_64_RELATIVE**

This is **almost entirely Solana-specific** and behaves differently based on whether the relocation target is in the .text section or not :

- **In .text**: Behaves like R_BPF_64_64 but ignores R_SYM(r_info) and converts physical addresses to virtual addresses by adding MM_PROGRAM_ADDR
- **Outside .text**: Performs a 64-bit write, adding MM_PROGRAM_ADDR to the implicit addend

```
[0x1000007388]> pd 3 @ 0x1000021420
            0x1000021420      18010000d0e402..    lddw r1, 0x100002e4d0 ; RELOC 64
            0x1000021430      7b1ad0ff00000000    stxdw [r10+0xffd0], r1
            0x1000021438      7a0af0ff00000000    stdw [r10+0xfff0], 0x0
[0x1000007388]> pd 1 @ 0x100002e4d0
            0x100002e4d0      .qword 0x000000100002c5e2                ; RELOC 64
[0x1000007388]> pxq 16 @ 0x100002e4d0
0x100002e4d0  0x000000100002c5e2  0x0000000000000011  ................
[0x1000007388]> px 0x11 @ 0x000000100002c5e2
- offset -    E2E3 E4E5 E6E7 E8E9 EAEB ECED EEEF F0F1  23456789ABCDEF01
0x100002c5e2  6361 7061 6369 7479 206f 7665 7266 6c6f  capacity overflo
0x100002c5f2  77                                       w
```

```
[0x1000007388]> pd 3 @ 0x1000021420
            0x1000021420      18010000d0e402..    lddw r1, 0x100002e4d0 ; RELOC 64
            0x1000021430      7b1ad0ff00000000    stxdw [r10+0xffd0], r1
            0x1000021438      7a0af0ff00000000    stdw [r10+0xfff0], 0x0
[0x1000007388]> pd 1 @ 0x100002e4d0
            0x100002e4d0      .qword 0x000000100002c5e2                ; RELOC 64
[0x1000007388]> pxq 16 @ 0x100002e4d0
0x100002e4d0  0x000000100002c5e2  0x0000000000000011  ................
[0x1000007388]> px 0x11 @ 0x000000100002c5e2
- offset -    E2E3 E4E5 E6E7 E8E9 EAEB ECED EEEF F0F1  23456789ABCDEF01
0x100002c5e2  6361 7061 6369 7479 206f 7665 7266 6c6f  capacity overflo
0x100002c5f2  77                                       w
```

3. **R_BPF_64_32**

Sets the 32-bit immediate field of a call instruction to either :

- **For local functions**: A Murmur3 hash of the function's PC address
- **For syscalls**: The syscall ID (also a Murmur3 hash of the symbol name)

```
9c2500000000000000   ldxq r5, [r2+0x0]
9f1600000000000000   stxq [r6+0x0], r1
bc3100000000000000   mov r1, r3
5501130001000000     jne r1, 0x1, 0x1000005d30
8510000ffffffff      syscall                    ;[2]; RELOC 32
bfa1000000000000     mov64 r1, r10
07010000f0000000     add64 r1, 0xf0
```

Syscalls

Syscalls in sBPF are implemented through a **registration and dispatch system** that bridges VM programs to native runtime functions. The implementation consists of three main components:

- Registration
- Invocation
- Execution

Syscall Registration

Syscalls are registered into a hash table during SVM initialization. Each syscall entry contains:

- A **Murmur3-32 hash** of the syscall name as the key
- A **function pointer** to the native implementation
- The **syscall name** string

The registration process conditionally registers syscalls based on feature flags. For example, sol_sha256 may be always registered, while sol_blake3 is only registered if the blake3_syscall_enabled feature is active.

```
REGISTER( "sol_log_",                          fd_vm_syscall_sol_log );
REGISTER( "sol_log_64_",                       fd_vm_syscall_sol_log_64 );
REGISTER( "sol_log_compute_units_",            fd_vm_syscall_sol_log_compute_units );
REGISTER( "sol_log_pubkey",                    fd_vm_syscall_sol_log_pubkey );
REGISTER( "sol_create_program_address",        fd_vm_syscall_sol_create_program_address );
REGISTER( "sol_try_find_program_address",      fd_vm_syscall_sol_try_find_program_address );
REGISTER( "sol_sha256",                        fd_vm_syscall_sol_sha256 );
REGISTER( "sol_keccak256",                     fd_vm_syscall_sol_keccak256 );
REGISTER( "sol_secp256k1_recover",             fd_vm_syscall_sol_secp256k1_recover );

if( enable_blake3_syscall )
  REGISTER( "sol_blake3",                      fd_vm_syscall_sol_blake3 );
```

```
int
fd_vm_syscall_register( fd_sbpf_syscalls_t *   syscalls,
                        char const *           name,
                        fd_sbpf_syscall_func_t func ) {
  if( FD_UNLIKELY( (!syscalls) | (!name) ) ) return FD_VM_ERR_INVAL;

  fd_sbpf_syscalls_t * syscall = fd_sbpf_syscalls_insert( map: syscalls, key: (ulong)fd_murmur3_32( data: name, sz: strlen( s: name ), seed: 0U ) );
  if( FD_UNLIKELY( !syscall ) ) return FD_VM_ERR_INVAL; /* name (or hash of name) already in map */

  syscall->func = func;
  syscall->name = name;

  return FD_VM_SUCCESS;
}
```

Firedancer: [src/flamenco/vm/syscall/fd_vm_syscall.c]

INVERSIVE LABS

```
REGISTER( "sol_log_",                          fd_vm_syscall_sol_log );
REGISTER( "sol_log_64_",                       fd_vm_syscall_sol_log_64 );
REGISTER( "sol_log_compute_units_",            fd_vm_syscall_sol_log_compute_units );
REGISTER( "sol_log_pubkey",                    fd_vm_syscall_sol_log_pubkey );
REGISTER( "sol_create_program_address",        fd_vm_syscall_sol_create_program_address );
REGISTER( "sol_try_find_program_address",      fd_vm_syscall_sol_try_find_program_address );
REGISTER( "sol_sha256",                        fd_vm_syscall_sol_sha256 );
REGISTER( "sol_keccak256",                     fd_vm_syscall_sol_keccak256 );
REGISTER( "sol_secp256k1_recover",             fd_vm_syscall_sol_secp256k1_recover );

if( enable_blake3_syscall )
  REGISTER( "sol_blake3",                      fd_vm_syscall_sol_blake3 );
```

```
int
fd_vm_syscall_register( fd_sbpf_syscalls_t *   syscalls,
                        char const *           name,
                        fd_sbpf_syscall_func_t func ) {
  if( FD_UNLIKELY( (!syscalls) | (!name) ) ) return FD_VM_ERR_INVAL;

  fd_sbpf_syscalls_t * syscall = fd_sbpf_syscalls_insert( map: syscalls, key: (ulong)fd_murmur3_32( data: name, sz: strlen( s: name ), seed: 0U ) );
  if( FD_UNLIKELY( !syscall ) ) return FD_VM_ERR_INVAL; /* name (or hash of name) already in map */

  syscall->func = func;
  syscall->name = name;

  return FD_VM_SUCCESS;
}
```

Firedancer: [src/flamenco/vm/syscall/fd_vm_syscall.c]

INVERSIVE LABS

Syscall Invocation

When a program executes a syscall instruction (opcode 0x85 for CALL_IMM or 0x95 for static syscalls), the VM interpreter handles it in two ways depending on the SBPF version:

- Dynamic Syscalls (sBPF v0-v2)
  For older SBPF versions, the syscall hash is looked up dynamically:
    - The imm field contains a hash value
    - The VM queries the syscalls hash table
    - If found, the syscall function pointer is retrieved
    - If not found, it's treated as a local function call

- Static Syscalls (sBPF v3+)
  For SBPF v3+, syscalls use direct indexing for better performance. (relocation is applied at compilation time)

Before Applying Relocations

```
79a190ff00000000   ldxdw r1, [r10+0xff90]
79a298ff00000000   ldxdw r2, [r10+0xff98]
85100000ffffffff   syscall
736a78ff00000000   stxb [r10+0xff78], r6
0706000037000000   add64 r6, 0x37
```

After Applying Relocations

```
79a190ff00000000   ldxdw r1, [r10+0xff90]
79a298ff00000000   ldxdw r2, [r10+0xff98]
85100000bd597520   syscall sol_log_
736a78ff00000000   stxb [r10+0xff78], r6
0706000037000000   add64 r6, 0x37
```

Before Applying Relocations

```
79a190ff00000000   ldxdw r1, [r10+0xff90]
79a298ff00000000   ldxdw r2, [r10+0xff98]
85100000ffffffff   syscall
736a78ff00000000   stxb [r10+0xff78], r6
0706000037000000   add64 r6, 0x37
```

After Applying Relocations

```
79a190ff00000000   ldxdw r1, [r10+0xff90]
79a298ff00000000   ldxdw r2, [r10+0xff98]
85100000bd597520   syscall sol_log_
736a78ff00000000   stxb [r10+0xff78], r6
0706000037000000   add64 r6, 0x37
```

```
static fd_murmur3_32_test_vector_t const fd_murmur3_32_test_vector[] = {
  [0]={ .hash=0xb6fc1a11U, .msg="abort",                          .sz=5UL,  .seed=0 },
  [1]={ .hash=0x686093bbU, .msg="sol_panic_",                     .sz=10UL, .seed=0 },
  [2]={ .hash=0x207559bdU, .msg="sol_log_",                       .sz=8UL,  .seed=0 },
  [3]={ .hash=0x5c2a3178U, .msg="sol_log_64_",                    .sz=11UL, .seed=0 },
  [4]={ .hash=0x52ba5096U, .msg="sol_log_compute_units_",         .sz=22UL, .seed=0 },
  [5]={ .hash=0x7ef088caU, .msg="sol_log_pubkey",                 .sz=14UL, .seed=0 },
  [6]={ .hash=0x9377323cU, .msg="sol_create_program_address",     .sz=26UL, .seed=0 },
  [7]={ .hash=0x48504a38U, .msg="sol_try_find_program_address",   .sz=28UL, .seed=0 },
```

Firedancer: [src/ballet/murmur3/test_murmur3.c]

INVERSIVE LABS

Before Applying Relocations

```
79a190ff00000000   ldxdw r1, [r10+0xff90]
79a298ff00000000   ldxdw r2, [r10+0xff98]
8510000ffffffff    syscall
736a78ff00000000   stxb [r10+0xff78], r6
0706000037000000   add64 r6, 0x37
```

After Applying Relocations

```
79a190ff00000000   ldxdw r1, [r10+0xff90]
79a298ff00000000   ldxdw r2, [r10+0xff98]
8510000bd597520    syscall sol_log_
736a78ff00000000   stxb [r10+0xff78], r6
0706000037000000   add64 r6, 0x37
```

```
static fd_murmur3_32_test_vector_t const fd_murmur3_32_test_vector[] = {
  [0]={ .hash=0xb6fc1a11U, .msg="abort",                          .sz=5UL,  .seed=0 },
  [1]={ .hash=0x686093bbU, .msg="sol_panic_",                     .sz=10UL, .seed=0 },
  [2]={ .hash=0x207559bdU, .msg="sol_log_",                       .sz=8UL,  .seed=0 },
  [3]={ .hash=0x5c2a3178U, .msg="sol_log_64_",                    .sz=11UL, .seed=0 },
  [4]={ .hash=0x52ba5096U, .msg="sol_log_compute_units_",         .sz=22UL, .seed=0 },
  [5]={ .hash=0x7ef088caU, .msg="sol_log_pubkey",                 .sz=14UL, .seed=0 },
  [6]={ .hash=0x9377323cU, .msg="sol_create_program_address",     .sz=26UL, .seed=0 },
  [7]={ .hash=0x48504a38U, .msg="sol_try_find_program_address",   .sz=28UL, .seed=0 },
```

Firedancer: [src/ballet/murmur3/test_murmur3.c]

INVERSIVE LABS

sBPF V0-V2

```
79a190ff00000000    ldxdw r1, [r10+0xff90]
79a298ff00000000    ldxdw r2, [r10+0xff98]
85100000bd597520    syscall sol_log_
736a78ff00000000    stxb [r10+0xff78], r6
0706000037000000    add64 r6, 0x37
```

sBPF V3

```
9ca1880000000000    ldxq r1, [r10+0x88]
9ca2900000000000    ldxq r2, [r10+0x90]
95000000bd597520    syscall sol_log_
2f6a700000000000    stxb [r10+0x70], r6
0406000037000000    add r6, 0x37
```

sBPF V0-V2

```
79a190ff00000000    ldxdw r1, [r10+0xff90]
79a298ff00000000    ldxdw r2, [r10+0xff98]
85100000bd597520    syscall sol_log_
736a78ff00000000    stxb [r10+0xff78], r6
0706000037000000    add64 r6, 0x37
```

sBPF V3

```
9ca1880000000000    ldxq r1, [r10+0x88]
9ca2900000000000    ldxq r2, [r10+0x90]
95000000bd597520    syscall sol_log_
2f6a700000000000    stxb [r10+0x70], r6
0406000037000000    add r6, 0x37
```

sBPF V0-V2

```
79a190ff00000000    ldxdw r1, [r10+0xff90]
79a298ff00000000    ldxdw r2, [r10+0xff98]
8510000000bd597520  syscall sol_log_
736a78ff00000000    stxb [r10+0xff78], r6
0706000037000000    add64 r6, 0x37
```

sBPF V3

```
9ca1880000000000    ldxq r1, [r10+0x88]
9ca2900000000000    ldxq r2, [r10+0x90]
9500000000bd597520  syscall sol_log_
2f6a700000000000    stxb [r10+0x70], r6
0406000037000000    add r6, 0x37
```

Syscall Execution

When a syscall executes:

- **Compute Unit Deduction**: Every syscall starts by deducting compute units The base cost is typically 100 CU, with additional costs for operations.

- **Memory Translation**: VM virtual addresses are translated to host addresses, performing bounds checking and permission validation.

- **Operation Execution**: The syscall performs its specific operation (e.g., hashing, memory operations, CPI).

- **Return Values**: Generally the syscall result is written to the value in register r0, although some syscalls implement multiple return values by writting to other registers aswell

4. Radare2 Support for Solana Programs

Key Problems Solved:

- Support for disassembling sBPF v0/v1/v2/v3
- Support for sBPF relocations
- Syscall Identification
  - Solana syscalls resolution by mapping hash values to readable names
- sBPF pseudo-decompilation (thanks @trufae! <3)
- Rust String Detection - Handles Rust's non-null-terminated strings by:
  - Finding LDDW or analogous instructions pointing to data segments
  - Detecting Rust String structures
  - Creating string xrefs, comments, and flags

INVERSIVE LABS

Demo

Moving Forward:

- We aim to keep contributing to radare2 and enhance the features of sBPF plugins to give back to the community
- Inlined stack string detection
- Highly improve pseudo decompilation
- Stay tuned for more updates at https://inversive.xyz

INVERSIVE LABS

5. References

- Firedancer: https://github.com/firedancer-io/firedancer
- rbpf: https://github.com/solana-labs/rbpf
- Agave: https://github.com/anza-xyz/agave
- Helius: https://www.helius.dev/blog/solana-executive-overview
- squads: https://squads.so/blog/solana-svm-sealevel-virtual-machine
- Solana Documentation: https://solana.com/docs
- Solana White-paper: https://solana.com/solana-whitepaper.pdf
- BlueShift sBPF Linker: https://github.com/blueshift-gg/sbpf-linker

INVERSIVE LABS