Lecture #9: Accounts, Gas, and Virtual Machines

COMS 4995-001:
The Science of Blockchains

URL: https://timroughgarden.org/s25/

Tim Roughgarden

The Execution Layer

Execution layer: keep state of the virtual machine up-to-date.

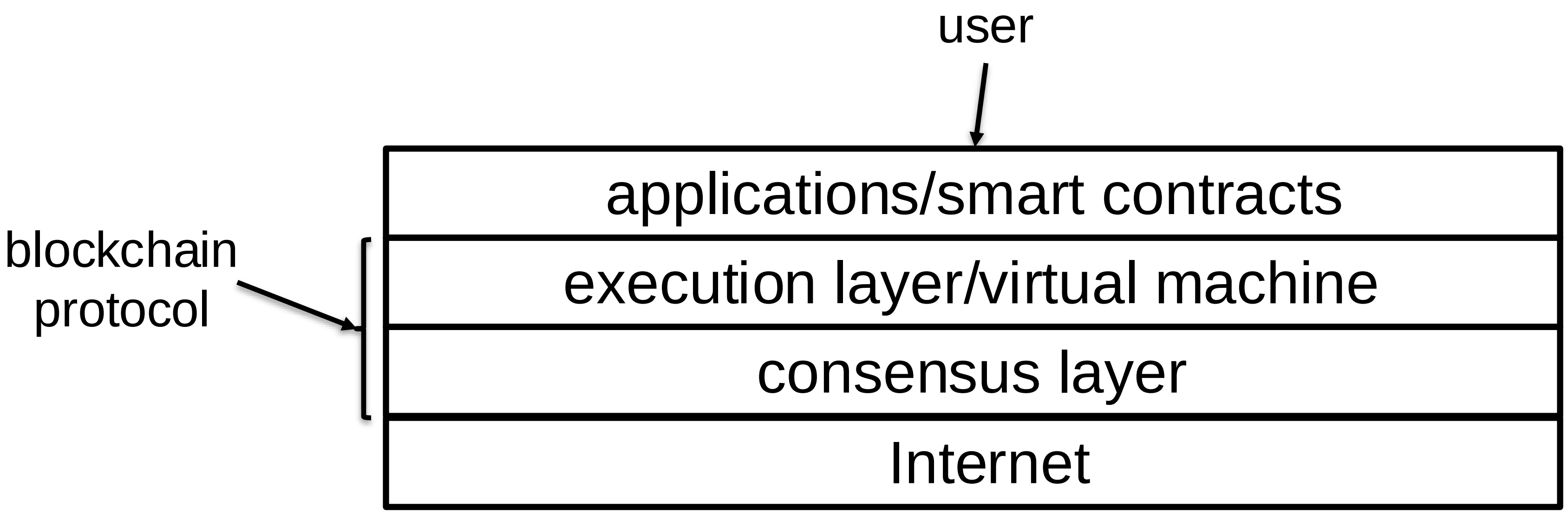
tx2	tx3	tx1	tx4

consensus transaction sequence

blockchain protocol
(execution layer)
[replicated at each
physical machine]

simulated (virtual) computer
[replicated at each physical
machine]

A Cartoon of Web3 (Refined)



Blockchain protocol:

- like an operating system, a blockchain protocol:
  - acts as a “master program” to coordinate all apps/smart contracts
  - provides a virtual machine to developers of applications
- “decentralized” like the Internet

Goals for Lecture #9

1. The account-base model (used in Ethereum, Solana, etc.).
   - explicit notion of account IDs and balances, programs as accounts
2. Transactions.
   - high-level: the basic unit of user intent
   - low-level: a snippet of VM code
3. Virtual machines.
   - how to execute transactions with arbitrary code
4. Metering computation.

How to Think About the Execution Layer

Questions: what are the possible “states” of the virtual machine?

- how are transactions described (both high-level and low-level)?
- what state transition results from executing a transaction?
- how does a validator represent state and carry out transitions?

How to Think About the Execution Layer

Questions: what are the possible “states” of the virtual machine?

- how are transactions described (both high-level and low-level)?
- what state transition results from executing a transaction?
- how does a validator represent state and carry out transitions?

Note: will now treat the consensus layer as a “black box,” consider a single validator processing a transaction sequence.

- – separation between consensus and execution varies with protocol

How to Think About the Execution Layer

Questions: what are the possible "states" of the virtual machine?

- how are transactions described (both high-level and low-level)?
- what state transition results from executing a transaction?
- how does a validator represent state and carry out transitions?

Note: will now treat the consensus layer as a "black box," consider a single validator processing a transaction sequence.

  – separation between consensus and execution varies with protocol

Next: account-based general-purpose execution layers.

Account-Based Execution Layers

Examples: Ethereum, Solana, Move (Diem/Aptos/Sui), etc.

Account-Based Execution Layers

Examples: Ethereum, Solana, Move (Diem/Aptos/Sui), etc.

Idea: “state” of an account-based protocol specified by:

- a set of current accounts (indexed by accountID, e.g. a pk)
  - generally, an account could correspond to a user or a program/contract

Account-Based Execution Layers

Examples: Ethereum, Solana, Move (Diem/Aptos/Sui), etc.

Idea: “state” of an account-based protocol specified by:

- a set of current accounts (indexed by accountID, e.g. a pk)
  - generally, an account could correspond to a user or a program/contract
- the state of each of these accounts, which could include:
  - balance in native cryptocurrency (ETH, SOL, etc.)
    - contracts can also have non-zero balances
  - arbitrary persistent and mutable data
  - VM code a.k.a. bytecode (typically immutable)

Account-Based Execution Layers

Idea: "state" of an account-based protocol specified by:

- a set of current accounts (indexed by accountID, e.g. a pk)
- the state of each of these accounts, which could include:
  - e.g., balance, data, and or bytecode

Account-Based Execution Layers

Idea: "state" of an account-based protocol specified by:

- a set of current accounts (indexed by accountID, e.g. a pk)
- the state of each of these accounts, which could include:
  - e.g., balance, data, and or bytecode

Example: Ethereum.

- distinguish user accounts ("EOAs") vs. contract accounts
  - EOA → no data, no code
  - contract account → both data and code (e.g., NFT contract)

Account-Based Execution Layers

Idea: "state" of an account-based protocol specified by:

- a set of current accounts (indexed by accountID, e.g. a pk)
- the state of each of these accounts, which could include:
  - e.g., balance, data, and or bytecode

Examples: Ethereum vs. Solana.

- Ethereum: distinguish user accounts vs. contract accounts
  - EOA → no data, no code; contract account → both data and code
- Solana: distinction between user vs. program accounts blurrier, program code and data stored in separate accounts

Account-Based Execution Layers

Idea: "state" of an account-based protocol specified by:

- a set of current accounts (indexed by accountID, e.g. a pk)
- the state of each of these accounts, which could include:
  - e.g., balance, data, and or bytecode

Examples: Ethereum vs. Solana.

Note: protocol does not fully prescribe how to represent state.

  - e.g., multiple execution clients in Ethereum, differ in many details
  - in practice, state typically includes metadata that strongly encourages a particular implementation (e.g., Merkle trees, see next week)

Transactions

Basic ingredients:

Transactions

Basic ingredients:

- a function call (to some contract, with some arguments)
- any necessary signatures
- priority fee
  - to encourage validators to include this transaction over others

Transactions

Basic ingredients:

- a function call (to some contract, with some arguments)
- any necessary signatures
- priority fee
  - to encourage validators to include this transaction over others

Extras:

- information about resources required by transaction
- defense against replay attacks

Example: Ethereum Transactions

- each transaction sent by a single EOA (i.e., only one signer)
- signature by sender required
  - can back out sender's public key (+ hence accountID) from signature

Example: Ethereum Transactions

- each transaction sent by a single EOA (i.e., only one signer)
- signature by sender required
  - can back out sender’s public key (+ hence accountID) from signature
- recipient (EOA or contract account, specified by accountID)

Example: Ethereum Transactions

- each transaction sent by a single EOA (i.e., only one signer)
- signature by sender required
  - can back out sender's public key (+ hence accountID) from signature
- recipient (EOA or contract account, specified by accountID)
- value (in ETH)
- data (e.g., function call + arguments)

Example: Ethereum Transactions

- each transaction sent by a single EOA (i.e., only one signer)
- signature by sender required
  - can back out sender's public key (+ hence accountID) from signature
- recipient (EOA or contract account, specified by accountID)
- value (in ETH)
- data (e.g., function call + arguments)
- "gas limit" (computation required by transaction (≈ "size"))
- "gas price" (priority fee to be paid per-unit-gas)

Example: Ethereum Transactions

- each transaction sent by a single EOA (i.e., only one signer)
- signature by sender required
  – can back out sender's public key (+ hence accountID) from signature
- recipient (EOA or contract account, specified by accountID)
- value (in ETH)
- data (e.g., function call + arguments)
- "gas limit" (computation required by transaction (≈ "size"))
- "gas price" (priority fee to be paid per-unit-gas)
- nonce (for replay attack protection)
  – for tx to be valid, needs to match nonce in sender's account

Example: USDC Transfer

Transaction Hash:	0x3b9ada8e0cbf69fb4aff9f40e9946bf3d5585f47ed03b403fd50f570870be168
Status:	Success
Block:	21879725 5 Block Confirmations
Timestamp:	53 secs ago (Feb-19-2025 10:09:47 AM UTC)
Transaction Action:	Transfer 101.4 ($101.39) USDC To 0x618a16ED7d4C39EFB5A6342C3972c896757a1b79
Sponsored:	
From:	0xc4a3Dcd48118D77bE44E7853bd5938F7448c7bD7
Interacted With (To):	0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48 (Circle: USDC Token)
ERC-20 Tokens Transferred:	All Transfers Net Transfers From 0xc4a3Dcd4...7448c7bD7 To 0x618a16ED...6757a1b79 For 101.4 ($101.39) USDC (USDC)
Value:	0 ETH ($0.00)
Transaction Fee:	0.00008286227098344 ETH ($0.23)
Gas Price:	1.442086164 Gwei (0.000000001442086164 ETH)
Gas Limit & Usage by Txn:	94,566 | 57,460 (60.76%)
Gas Fees:	Base: 0.732318081 Gwei | Max: 1.6 Gwei | Max Priority: 0.709768083 Gwei
Burnt & Txn Savings Fees:	Burnt: 0.00004207899693426 ETH ($0.11) Txn Savings: 0.00000907372901656 ETH ($0.02)
Other Attributes:	Txn Type: 2 (EIP-1559) Nonce: 0 Position In Block: 165
Input Data:	(see below)

```
Function: transfer(address to,uint256 value)

MethodID: 0xa9059cbb
[0]:  000000000000000000000000618a16ed7d4c39efb5a6342c3972c896757a1b79
[1]:  00000000000000000000000000000000000000000000000000000000060b3dc0
```

Example: Solana Transactions

- list of instructions (one instruction ≈ function call + arguments)

Example: Solana Transactions

- list of instructions (one instruction ≈ function call + arguments)
- list of accounts to be accessed
  - read vs. read-write access, whether or not signature is required

Example: Solana Transactions

- list of instructions (one instruction ≈ function call + arguments)
- list of accounts to be accessed
  - read vs. read-write access, whether or not signature is required
- any signatures required
- priority fee (paid per-VM-instruction, on top of “base fee”)

Example: Solana Transactions

- list of instructions (one instruction ≈ function call + arguments)
- list of accounts to be accessed
  - read vs. read-write access, whether or not signature is required
- any signatures required
- priority fee (paid per-VM-instruction, on top of "base fee")
- timestamp (for replay attack protection)
  - for tx to be valid, must have recent timestamp + not be a duplicate

Signature	24GQPNtGKpE4GJ7dKpZKR2frMW1YVKyTaFbsofTo2HTvwZ8c2dXvVUnNab8puja7Y8KzUmpq1HJmFYsPfEpsitb4
Result	Success
Timestamp	Feb 19, 2025 at 05:19:07 Eastern Standard Time
Confirmation Status	FINALIZED
Confirmations	MAX
Slot	321,667,953
Recent Blockhash	BjPWZH6Lo4CXe8pwV3ErXphqcfG4kitcQPhhKUu3Ve2M
Fee (SOL)	◎0.000040001
Compute units consumed	65,011

#1 Compute Budget Program Instruction — Collapse

Program	Compute Budget Program Owned by Native Loader. Balance is 0.000000001 SOL.
Instruction Data (Hex)	02 30 57 05 00

#2 Compute Budget Program Instruction — Expand

#3 Associated Token Program Instruction — Expand

#4 Associated Token Program Instruction — Expand

#5 System Program Instruction — Expand

#6 Token Program Instruction — Expand

#7 Unknown Program (PhoeNiXZ8ByJGLkxNfZRnkUfjvmuYqLR89jjFHGqdXY) Instruction — Expand

#8 Token Program Instruction — Expand

Account List (14)

Collapse

Account #1 Signer Writable	2UD482Pweo5PTw6sQiwesc1GBNj2zdheKzMYTFMQbSvn Owned by System Program. Balance is 6.192685124 SOL.
Account #2 Writable	3HSYXeGc3LjEPCuzoNDjQN37F1ebsSiR4CqXVqQCdekZ Owned by Token Program. Balance is 0.00344699 SOL.
Account #3 Writable	3SSMjooESkRFjFP3uy1fSzrYi3ofMB4F6hc4GtXN9riL Account doesn't exist
Account #4 Writable	4DoNfFBfF7UokCC2FQzriy7yHK6DY6NVdYpuekQ5pRgg Owned by PhoeNiXZ8ByJGLkxNfZRnkUfjvmuYqLR89jjFHGqdXY. Balance is 12.09636746 SOL.
Account #5 Writable	6uyPZfSdJ5rAe9ZVYC4mNo6oPqjjvUgxMdtz9P6r8wqC Owned by Token Program. Balance is 0.00203928 SOL.
Account #6 Writable	8g4Z9d6PqGkgH31tMW6FwxGhwYJrXpxZHQrkikpLJKrG Owned by Token Program. Balance is 9,441.591547434 SOL.
Account #7	System Program Owned by Native Loader. Balance is 0.000000001 SOL.
Account #8	7aDTsspkQNGKmrexAN7FLx9oxU3iPczSSvHNggyuqYkR Account doesn't exist
Account #9	Associated Token Program Owned by BPF Loader 2. Balance is 0.7319136 SOL.
Account #10	Compute Budget Program Owned by Native Loader. Balance is 0.000000001 SOL.
Account #11	EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v Owned by Token Program. Balance is 376.645348171 SOL.
Account #12	PhoeNiXZ8ByJGLkxNfZRnkUfjvmuYqLR89jjFHGqdXY Owned by BPF Upgradeable Loader. Balance is 0.00114144 SOL.
Account #13	So11111111111111111111111111111111111111112 Owned by Token Program. Balance is 959.143176713 SOL.
Account #14	Token Program Owned by BPF Loader 2. Balance is 0.93408768 SOL.

Mental Model for General Transactions

Special case: in Ethereum, an EOA → EOA transaction (necessarily an ETH transfer) processed "natively."

- just modify the balances of the two EOA accounts directly
- ≈ processed directly by "operating system," not a "program"

Mental Model for General Transactions

Special case: in Ethereum, an EOA → EOA transaction (necessarily an ETH transfer) processed "natively."

- just modify the balances of the two EOA accounts directly
- ≈ processed directly by "operating system," not a "program"
- in Solana, SOL transfers still involve a function call (to a "pre-installed" program)

Mental Model for General Transactions

General transaction: can trigger arbitrary code.

»

Mental Model for General Transactions

General transaction: can trigger arbitrary code.

»

Analogy: Java and the Java Virtual Machine (JVM).

- contracts/programs ≈ objects, reactive to function calls
- example: crowdfunding a la Kickstarter

Mental Model for General Transactions

General transaction: can trigger arbitrary code.

»

Analogy: Java and the Java Virtual Machine (JVM).

- contracts/programs ≈ objects, reactive to function calls
- example: crowdfunding a la Kickstarter

- Step 1: developers write (high-level) Java code.

Mental Model for General Transactions

```
class GoodArithmetic {
        byte addOneAndOne() {
                byte a = 1;
                byte b = 1;
                byte c = (byte) (a + b);
                return c;
        }
}
```

Mental Model for General Transactions

General transaction: can trigger arbitrary code.

»

Analogy: Java and the Java Virtual Machine (JVM).

– contracts/programs ≈ objects, reactive to function calls
– example: crowdfunding a la Kickstarter

- Step 1: developers write (high-level) Java code.
- Step 2: Java code compiled down to bytecode.
  – lower-level, but still hardware-independent

Mental Model for General Transactions

```java
class GoodArithmetic {
        byte addOneAndOne() {
                byte a = 1;
                byte b = 1;
                byte c = (byte) (a + b);
                return c;
        }
}
```

```
iconst_1          // Push int constant 1.
istore_1          // Pop into local variable 1, which is a: byte a = 1;
iconst_1          // Push int constant 1 again.
istore_2          // Pop into local variable 2, which is b: byte b = 1;
iload_1           // Push a (a is already stored as an int in local variable 1).
iload_2           // Push b (b is already stored as an int in local variable 2).
iadd              // Perform addition. Top of stack is now (a + b), an int.
int2byte          // Convert int result to byte (result still occupies 32 bits).
istore_3          // Pop into local variable 3, which is byte c: byte c = (byte) (a + b);
iload_3           // Push the value of c so it can be returned.
ireturn           // Proudly return the result of the addition: return c;
```

Mental Model for General Transactions

General transaction: can trigger arbitrary code.

»

Analogy: Java and the Java Virtual Machine (JVM).

- contracts/programs ≈ objects, reactive to function calls
- example: crowdfunding a la Kickstarter

- Step 1: developers write (high-level) Java code.
- Step 2: Java code compiled down to bytecode.
  - lower-level, but still hardware-independent
- Step 3: (optional) compilation of bytecode to (hardware-dependent) machine code. [hybrid: JIT compilation at runtime]

Mental Model for General Transactions

```java
class GoodArithmetic {
        byte addOneAndOne() {
                byte a = 1;
                byte b = 1;
                byte c = (byte) (a + b);
                return c;
        }
}
```

```
IMM   R0,   0x80
LOAD  R0,   R0
IMM   R1,   0x84
LOAD  R1,   R1
IMM   R2,   0x0
IMM   R3,   0x4
IMM   R4,   0x0
IMM   R5,   0x1
STORE R0,   R2
ADD   R0,   R0, R3
ADD   R4,   R4, R5
BNE   0x20, R4, R1
```

```
iconst_1        // Push int constant 1.
istore_1        // Pop into local variable 1, which is a: byte a = 1;
iconst_1        // Push int constant 1 again.
istore_2        // Pop into local variable 2, which is b: byte b = 1;
iload_1         // Push a (a is already stored as an int in local variable 1).
iload_2         // Push b (b is already stored as an int in local variable 2).
iadd            // Perform addition. Top of stack is now (a + b), an int.
int2byte        // Convert int result to byte (result still occupies 32 bits).
istore_3        // Pop into local variable 3, which is byte c: byte c = (byte) (a + b);
iload_3         // Push the value of c so it can be returned.
ireturn         // Proudly return the result of the addition: return c;
```

```
0x 60 00 00 80
0x A4 00 00 00
0x 60 01 00 84
0x A4 01 01 00
0x 60 02 00 00
0x 60 03 00 04
0x 60 04 00 00
0x 60 05 00 01
0x 08 00 00 02
0x 20 00 00 03
0x 20 04 04 05
0x 11 20 04 01
```

Example: Transactions in Ethereum

General transaction: can trigger arbitrary code.

»

Example: Transactions in Ethereum

General transaction: can trigger arbitrary code.

»

Step 1: developers write (high-level) Solidity code.

- Solidity developed specifically for Ethereum

Example: Transactions in Ethereum

```
contract ERC20Token {
    string public name;
    string public symbol;
    uint8 public decimals = 18;
    uint256 public totalSupply;

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    constructor(uint256 initialSupply, string memory tokenName, string memory tokenSymbol) {
        totalSupply = initialSupply * 10 ** uint256(decimals);
        balanceOf[msg.sender] = totalSupply;
        name = tokenName;
        symbol = tokenSymbol;
    }

    // Additional functions...
}
```

Example: Transactions in Ethereum

General transaction: can trigger arbitrary code.

»

Step 1: developers write (high-level) Solidity code.

- Solidity developed specifically for Ethereum

Step 2: Solidity code compiled down to EVM bytecode.

- stack-based, like JVM bytecode

Example: Transactions in Ethereum

General transaction: can trigger arbitrary code.

»

Step 1: developers write (high-level) Solidity code.

- Solidity developed specifically for Ethereum

Step 2: Solidity code compiled down to EVM bytecode.

- stack-based, like JVM bytecode

```
PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x18 PUSH1 0x0 SSTORE CALLVALUE
ISZERO PUSH1 0x13 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST JUMPDEST PUSH1
0x36 DUP1 PUSH1 0x21 PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1
0x60 PUSH1 0x40 MSTORE JUMPDEST PUSH1 0x0 DUP1 REVERT STOP LOG1 PUSH6
0x627A7A723058 KECCAK256 SLT 0xc9 0xbd STOP ISZERO 0x2f LOG1 0xc4
DUP1 0xf6 DUP3 PUSH32
0x81515BB19C3E63BF7ED9FFBB5FDA0265983AC7980029000000000000000000000
```

Example: Transactions in Ethereum

General transaction: can trigger arbitrary code.

»

Step 1: developers write (high-level) Solidity code.

– Solidity developed specifically for Ethereum

Step 2: Solidity code compiled down to EVM bytecode.

At runtime: in response to a transaction’s function call:

Example: Transactions in Ethereum

General transaction: can trigger arbitrary code.

»

Step 1: developers write (high-level) Solidity code.

- Solidity developed specifically for Ethereum

Step 2: Solidity code compiled down to EVM bytecode.

At runtime: in response to a transaction's function call:

- validator loads relevant bytecode into read-only memory
- allocates program counter (PC), stack memory, heap memory
- carries out bytecode instructions 1-by-1
- updates local memory and Ethereum's global state as needed

Example: Transactions in Solana

Step 1: developers write high-level code (usually Rust).

src › bin › c02p01.rs › ...

```rust
use std::cell::RefCell;
use std::collections::hash_map::DefaultHasher;
use std::collections::HashSet;
use std::fmt::Display;
use std::hash::Hasher;
use std::rc::Rc;

type NodeRef<T> = Rc<RefCell<Node<T>>>;

struct LinkedList<T> {
    head: Option<NodeRef<T>>,
}

struct Node<T> {
    data: T,
    next: Option<NodeRef<T>>,
    prev: Option<NodeRef<T>>,
}

struct Iter<T> {
    next: Option<NodeRef<T>>,
}

impl<T> Node<T> {
    fn tail(node: &NodeRef<T>) -> Option<NodeRef<T>> {
        if let Some(cur) = node.borrow().next.as_ref().cloned() {
            return Node::tail(&cur.clone());
        }
        Some(node.clone())
    }
```

Example: Transactions in Solana

Step 1: developers write high-level code (usually Rust).

Step 2: High-level code compiled down to Solana bytecode.

- minor variant of eBPF bytecode [register-based virtual machine]

```
0000000000000000 <detect_execve>:
       0:   r1 = 0x1c050444
       1:   *(u32 *)(r10 - 0x8) = r1
       2:   r1 = 0x954094701340819 ll
       4:   *(u64 *)(r10 - 0x10) = r1
       5:   r1 = 0x10523251403e5713 ll
       7:   *(u64 *)(r10 - 0x18) = r1
       8:   r1 = 0x43075a150e130d0b ll
      10:   *(u64 *)(r10 - 0x20) = r1
      11:   r1 = 0x0

0000000000000060 <LBB0_1>:
      12:   r2 = 0x0 ll
      14:   r2 += r1
      15:   r2 = *(u8 *)(r2 + 0x0)
      16:   r3 = r10
      17:   r3 += -0x20
      18:   r3 += r1
      19:   r4 = *(u8 *)(r3 + 0x0)
      20:   r2 ^= r4
      21:   *(u8 *)(r3 + 0x0) = r2
      22:   r1 += 0x1
      23:   if r1 == 0x1c goto +0x1 <LBB0_2>
      24:   goto -0xd <LBB0_1>

00000000000000c8 <LBB0_2>:
      25:   r3 = r10
      26:   r3 += -0x20
      27:   r1 = 0x1c ll
      29:   r2 = 0x4
      30:   call 0x6
      31:   r0 = 0x1
      32:   exit
```

Example: Transactions in Solana

Step 1: developers write high-level code (usually Rust).

Step 2: High-level code compiled down to Solana bytecode.

- minor variant of eBPF bytecode [register-based virtual machine]

At runtime: in response to a transaction's function call:

- validator executes corresponding Solana bytecode
  - either in software or via JIT compilation to machine code

Example: Transactions in Solana

Step 1: developers write high-level code (usually Rust).

Step 2: High-level code compiled down to Solana bytecode.

- minor variant of eBPF bytecode [register-based virtual machine]

At runtime: in response to a transaction's function call:

- validator executes corresponding Solana bytecode
  - either in software or via JIT compilation to machine code

Atomic transactions: (also in Ethereum) if transaction fails to complete, gets rolled back. (i.e., as if never executed)

Metering Computation

Question: if programs can be arbitrary code, what about the halting problem? [could a tx force an infinite loop in the VM?]

Metering Computation

Question: if programs can be arbitrary code, what about the halting problem? [could a tx force an infinite loop in the VM?]

Solution: associate a cost with each VM instruction, paid by user.

Metering Computation

Question: if programs can be arbitrary code, what about the halting problem? [could a tx force an infinite loop in the VM?]

Solution: associate a cost with each VM instruction, paid by user.

Example: in Ethereum:

Metering Computation

Question: if programs can be arbitrary code, what about the halting problem? [could a tx force an infinite loop in the VM?]

Solution: associate a cost with each VM instruction, paid by user.

Example: in Ethereum:

- associate an amount of “gas” with each EVM opcode
  - EVM opcodes = instruction set for VM code in Ethereum’s VM
  - add two numbers = 3 units of gas; evaluate SHA-256 = 30 units

Metering Computation

Question: if programs can be arbitrary code, what about the halting problem? [could a tx force an infinite loop in the VM?]

Solution: associate a cost with each VM instruction, paid by user.

Example: in Ethereum:

- associate an amount of "gas" with each EVM opcode
  - EVM opcodes = instruction set for VM code in Ethereum's VM
  - add two numbers = 3 units of gas; evaluate SHA-256 = 30 units
- user prepays for gas (recall "gas limit" and "gas price" fields)
- run out of gas mid-execution ➔ tx aborted and rolled back