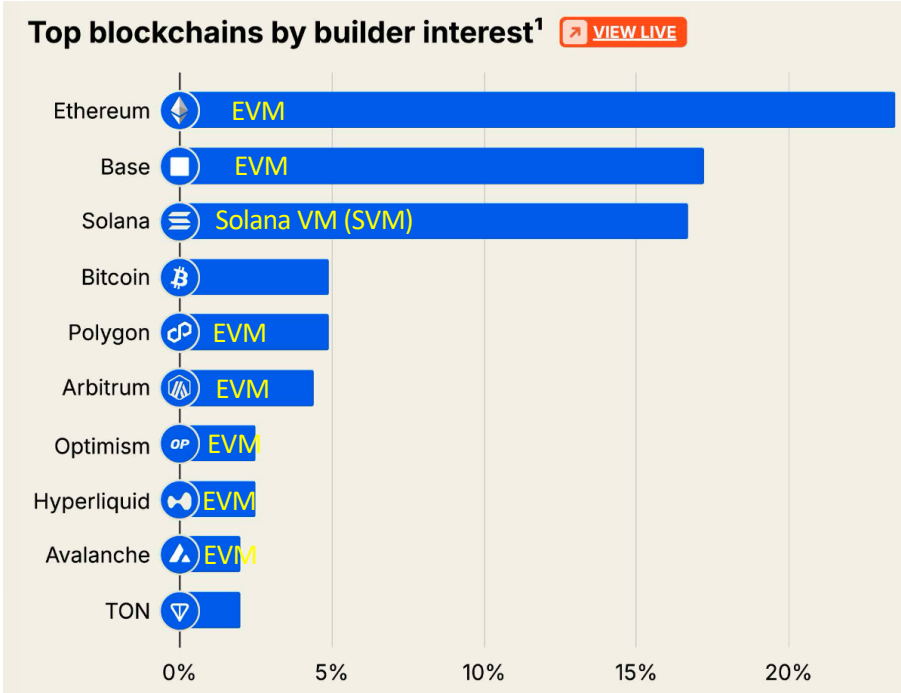
CS251 Fall 2025

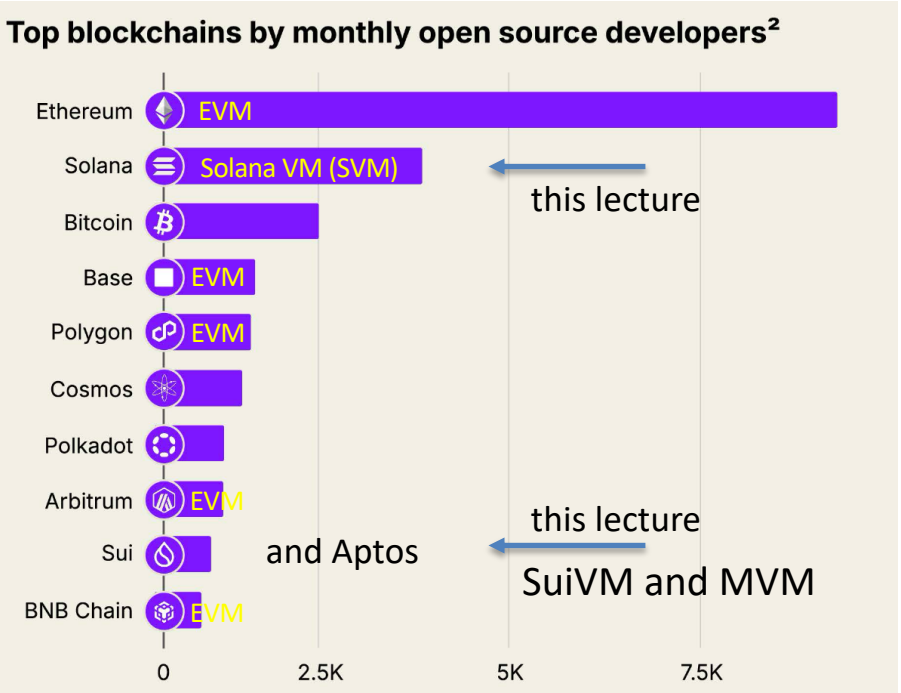
(cs251.stanford.edu)

Other layer-1 architectures: Solana, Sui, Aptos

Dan Boneh

This lecture: non-EVM chains





Key differences from the EVM

Recall: the EVM runs a **single thread** for the entire world

- transactions execute in a sequential order

Monad an EVM compatible chain, 10,000 Tx/sec !

- A re-implementation of the EVM that **optimistically executes transactions in parallel.**
- It re-executes transactions for which a conflict is detected
- Consensus: 0.400 sec. block time, 0.8 sec. to block finality

Types of Parallelism

A different approach: design a new VM for parallel execution

- Solana, Aptos, Sui, Avalanche, Flow, …

Solana: sealevel execution engine

- Every transaction explicitly lists the accounts that is touches
- None-overlapping transactions are executed in parallel

Aptos: Block-STM

- Optimistically execute transactions in parallel
- Re-execute sequentially if a conflict is discovered post-execution

Sui: Mysticeti consensus

- OwnedObject Tx (fast path) vs. SharedObject Tx (slow path)

Consensus block finality times

- **Ethereum**: 12 sec. slot, **12.8 minutes** to finalize an epoch

- **Solana**:
  - Today: 0.4 sec. block time, **12.8 sec.** to full finality
  - Coming Alpenglow consensus: **0.13** sec full finality

- **Aptos/Sui**: **sub-second** finality per block

Solana: a crash course

https://solana.com/docs

Mainnet launched on Mar. 2020 (genesis)

Node design

Ethereum philosophy: anyone can run an Ethereum node

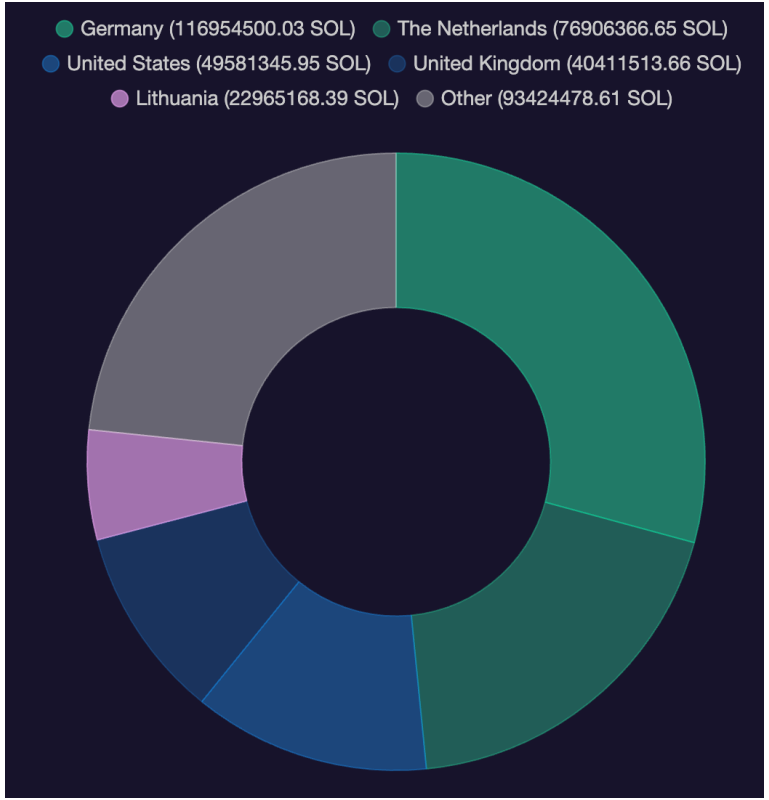
- Low-end and high-end machines ⇒ high resiliency

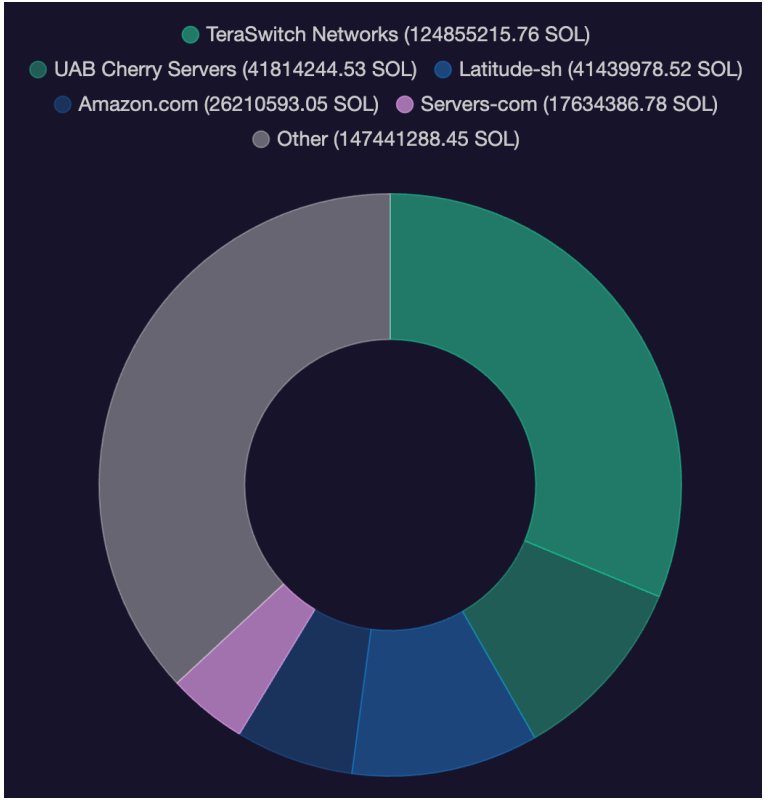
Solana philosophy: high-end servers, multi-core systems

⇒ lower network delays, faster execution time

- 971 (staked) validators
- Top 20 validators hold >33% of stake (for Ethereum this is 3)
  - can halt chain or censor transactions

Validator distribution



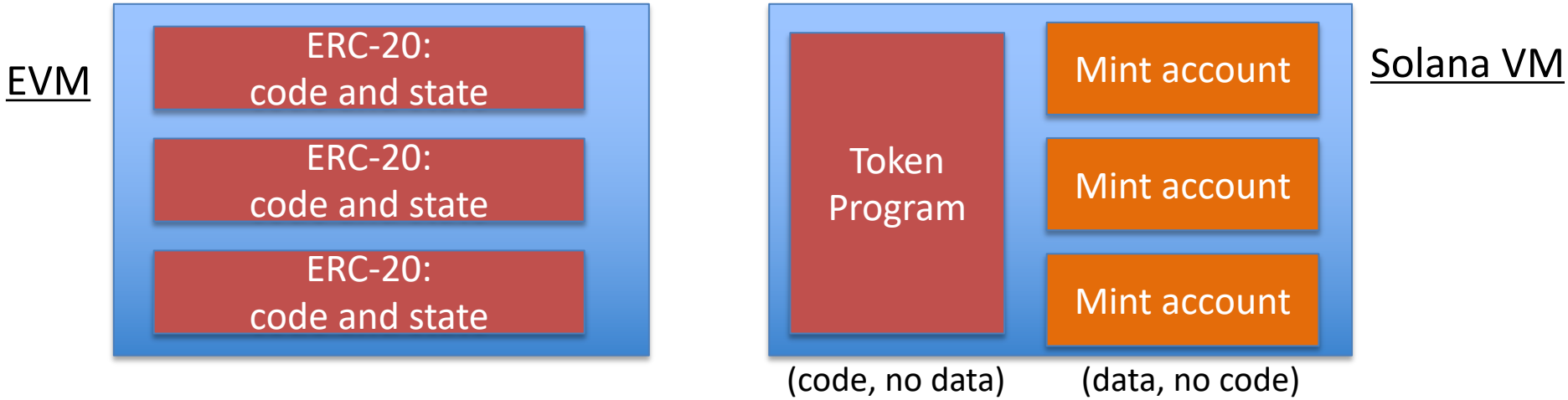


Solana account model

Ethereum: smart contracts carry code *and* state (the storage array)

Solana: programs carry code but are stateless !

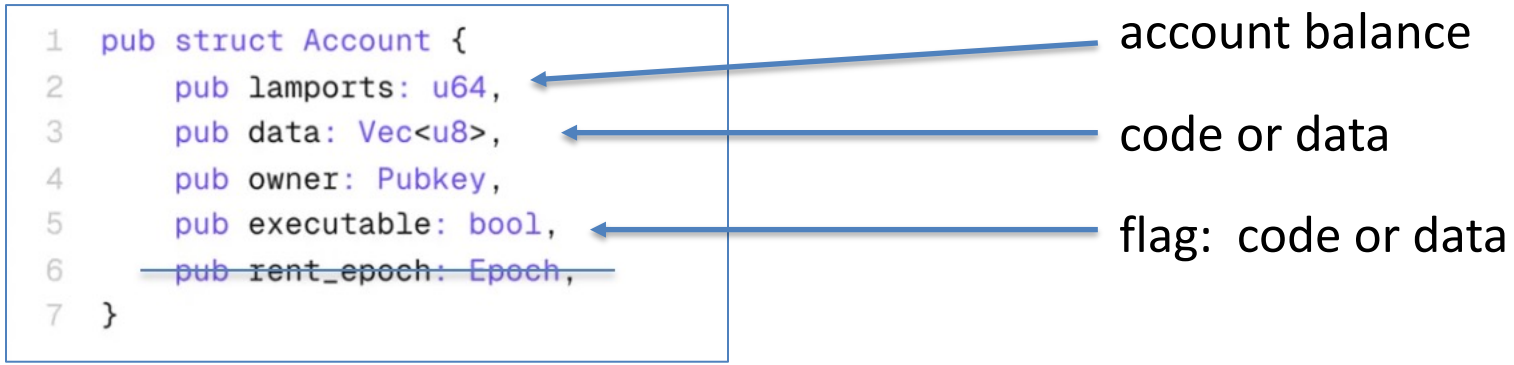
- state must be passed to a program to operate on.



Solana account model

A Solana account contains either:

- **State**: data that is meant to be read from and persisted. (balances, user data, etc.)
- **Executable Program**: code (sBPF byte code)



Lamports are the smallest unit of SOL. 1 SOL = 10^9 lamports.

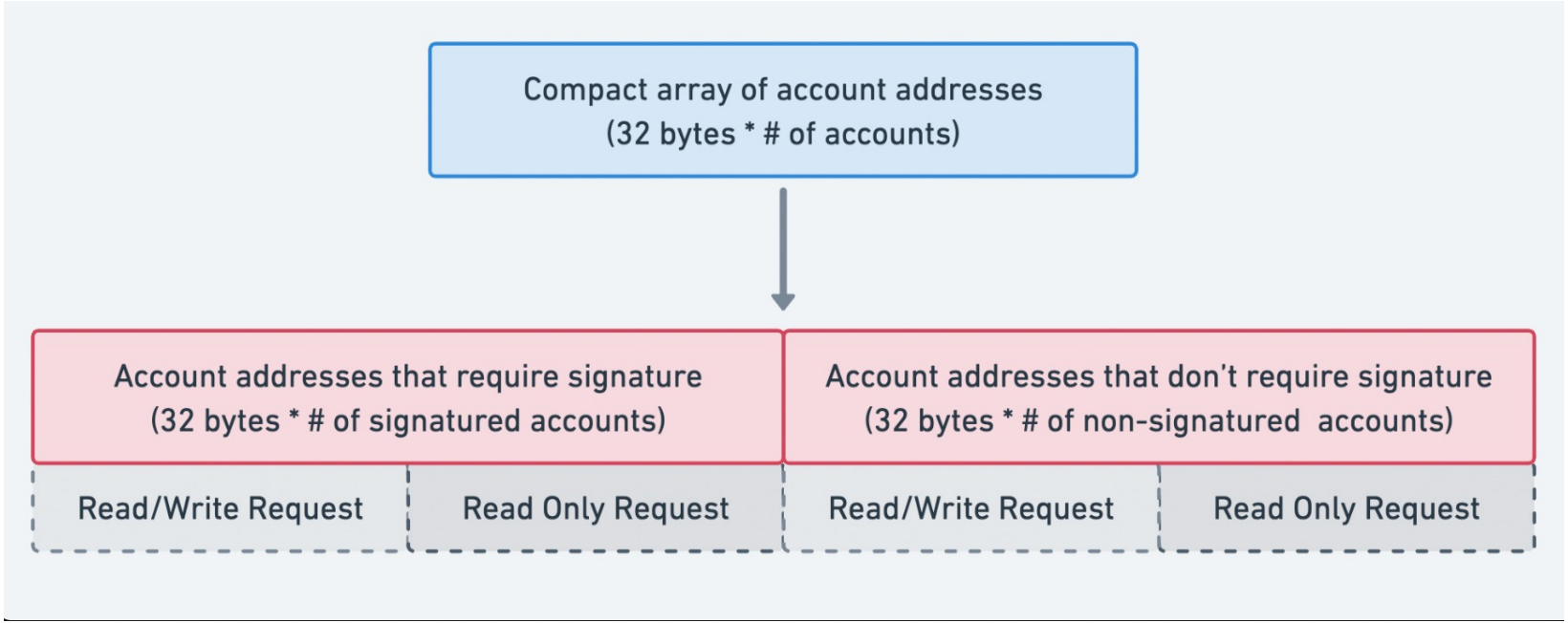
A Solana transaction

Four parts:

- An array of **accounts** to read or write from
- One or more **signatures** to authorize transaction
- One or more **instructions**
    - each instruction can be a function call to a program
    - call depth is ≤4 per instruction (no reentrency problems)
- A recent **blockhash** or nonce:
    - transaction is only valid for 150 blocks after blockhash

Solana

Every transaction includes a list of all accessed accounts:



Solana mempool?

There is no mempool!

- Leader schedule is pre-determined in advance
  - Chosen at random (by stake) from the set of validators
- All Tx are forwarded to the leader, who builds the block
- Left-over Tx are forwarded to the next leader

No need for a gossip protocol to replicate
the mempool across all validators

Alpenglow consensus

Validators vote on a proposed block from current leader:

- **Fast path**: if 80% of stake is responsive, block is finalized in a single round.
- **Slow path**: if 60% of stake is responsive, block is finalized in two rounds.
- Otherwise, no block: slot is empty.

>20% of stake is needed to violate safety

>40% of stake is needed to violate liveness

Aptos: a crash course

Move VM vs. EVM

Mainnet launched on Oct. 2022 (genesis)

Data is stored in a distributed way on Aptos

Recall: in an EVM **ERC-20** contract

- The balances of all holders stored in a single mapping in contract

  ⇒ every transaction touches global ERC-20 state

In **Aptos**:

- every user's balance is stored in an object owned by that user

  ⇒ similar to Solana's account model

- Better performance ⇒ can parallelize transaction execution

  a user transaction only touches that user's data

Signer as source of access control

On EVM, **tx.origin** is just the address of the initiating EOA.

- Recall: address of EOA is hash(PublicKey). Same in Bitcoin and Solana.

On Aptos an address stores its associated authentication key as a data field

⇒ can change authentication key **without changing the address**.

⇒ easier post-quantum transition

On Aptos there's a special signer type:

An entry function can require multiple signers in order to be called.

function that can be the target of a Tx

Static dispatch vs. dynamic dispatch

Cross contract call on EVM is flexible, even if you don't have the ABI or the source code, you can still make the call.

Aptos is static by design: you need (i) the bytecode, or (ii) source code, or (iii) mock interface, to make the call.

More recently: **function value**

- functions can be passed around as arguments

The MOVE language

Ready-made Smart Contract Frameworks

Objects

A framework to organize your data as extensible objects

Digital Assets

Allow to manage different form of assets on top of Aptos Objects

Smart Data Structures

Collection types which automatically adapt to your usage scenarios

Multisigner Transactions

The ability to authorize a transaction by more than one signer

Aggregators

Making sequential workloads execute in parallel

Safe Randomness

Supports safe, unbiased generation of random values

Keyless Accounts

Supports any OIDC (OpenID Connect) provider (Google, Facebook, Github, Apple, ...)

Fungible Asset Dynamic Dispatch

Allow to register transfer functions for dynamic dispatch

Your First Contract

Move syntax is similar to Rust

Everything is a **module**

```
module my_addr::hello_blockchain {
    use std::string::String;

    struct Message has key, copy, drop {
        message: String
    }

    public entry fun create(
        caller: &signer,
        msg: String
    ) {
        move_to(caller, Message {message: msg})
    }

    // ..
}
```

Two types of Move Programs

```
script {
        fun main() {
            ..
        }
}
```

Script

Executes a series of commands in a single transaction, without being stored on-chain

```
module my_addr::example {
    struct Foo {}

    public fun do_something() {
        ..
    }

    ..
}
```

Module

Stored on-chain.
Code reuse and composability

Module Definition

Modules are defined by

- Published address in global storage (can be a string or a literal address)
- Module name

Items scoped to modules

```
module 0x42::hello_blockchain {
    use std::string::String;

    struct Message has key, copy, drop {
        message: String
    }

    public entry fun create(
        caller: &signer,
        msg: String
    ) {
        move_to(caller, Message {message: msg})
    }

    // ..
}
```

Import Definition

Imports are defined as `use`

`use` lets you import other libraries, structs, or functions

```
module my_addr::hello_blockchain {
    use std::string::String;

    struct Message has key, copy, drop {
        message: String
    }

    public entry fun create(
        caller: &signer,
        msg: String
    ) {
        move_to(caller, Message {message: msg})
    }

    // ..
}
```

Struct Definition

Structs are custom data
- Similar to most languages

Identified by:
- Module address
- Module name
- Struct name

Scoped to a module

```
module my_addr::hello_blockchain {
    use std::string::String;

    struct Message has key, copy, drop {
        message: String
    }

    public entry fun create(
        caller: &signer,
        msg: String
    ) {
        move_to(caller, Message {message: msg})
    }

    // ..
}
```

Struct can have four abilities: copy, drop, store, key

drop: allows the struct to be dropped when it goes out of scope. None droppable structs must be explicitly destroyed at the end of a function. Tokens are usually not droppable.

key: allows the struct to be used as a top-level key in the global storage (automatically has store)

copy: allows the struct to be copied. Non-copyable structs change ownership when moved. Tokens are usually not copyable.

store: allows the struct to be stored in the global storage (also as part of another Struct)

```
struct Person has copy, drop, store, key {
    name: string,
    age: u64,
}
```

Using abilities

By combining abilities (or lack thereof), it is possible to achieve properties underlying safe use of resources

Example: a **Coin** struct that is storable but cannot be arbitrarily duplicated or destroyed!

```
struct Coin has store {
    name: string,
    age: u64,
}
```

Struct Definition: Inner Types

Structs allow for any inner types

- Can be other store structs

Inner types can only be accessed *in the module*

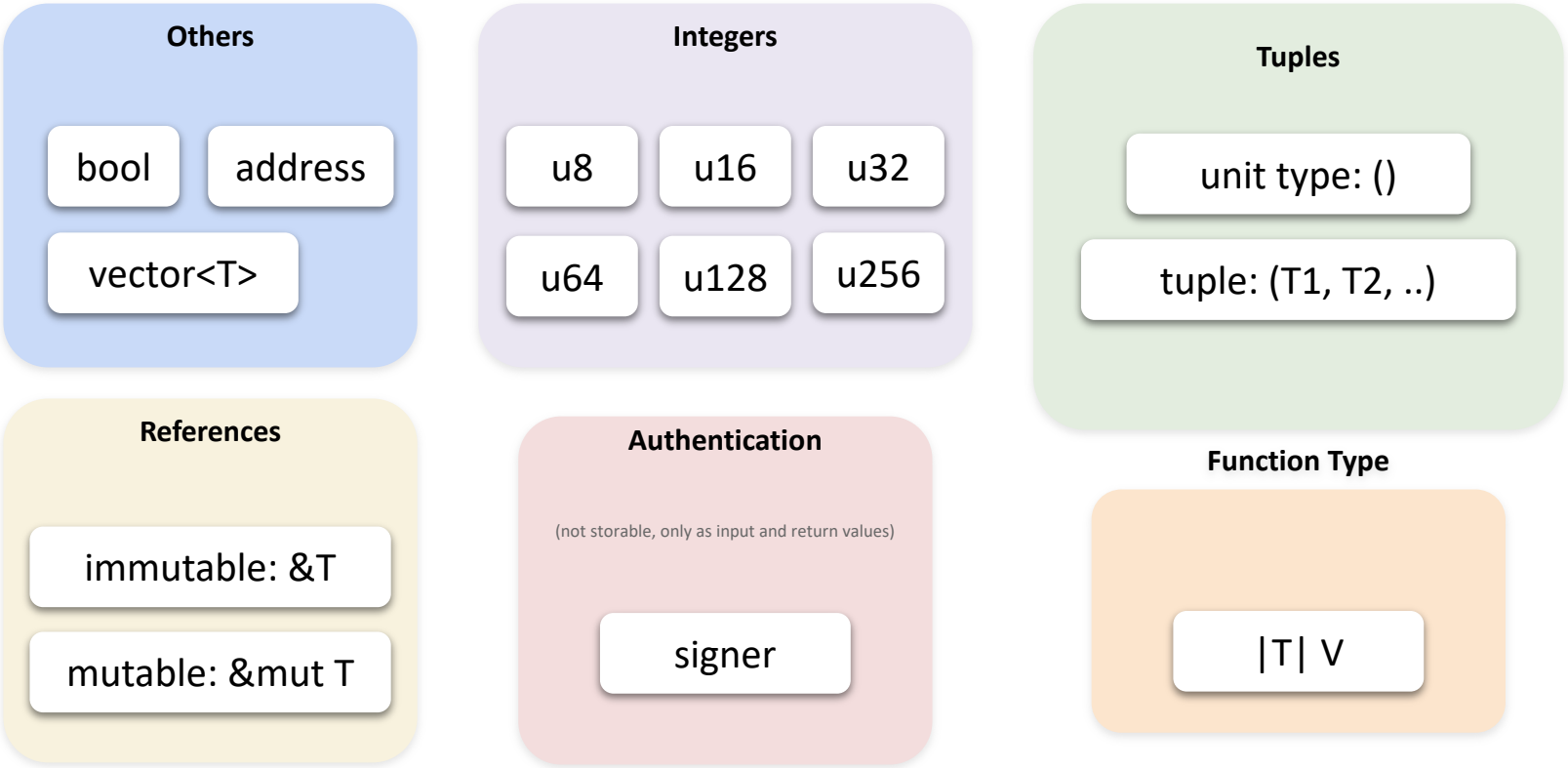
```
module my_addr::hello_blockchain {
    use std::string::String;

    struct Message has key, copy, drop {
        message: String
    }

    public entry fun create(
        caller: &signer,
        msg: String
    ) {
        move_to(caller, Message {message: msg})
    }

    // ..
}
```

Move Primitive Types



Function Definitions: Signature

Visibility can be

- **public**: called from any module
- **friend**: can only be called from modules declared as friends
- **private** (no keyword): can only be called from inside module

entry: callable by a transaction

inline: inserted into calling code at compile time

```
module my_addr::hello_blockchain {
    use std::string::String;

    struct Message has key, copy, drop {
        message: String
    }

    public entry fun create(
        caller: &signer,
        msg: String
    ) {
        move_to(caller, Message {message: msg})
    }
}
```

Function Signature: Arguments

Functions can take any type as input, including structs

Entry functions

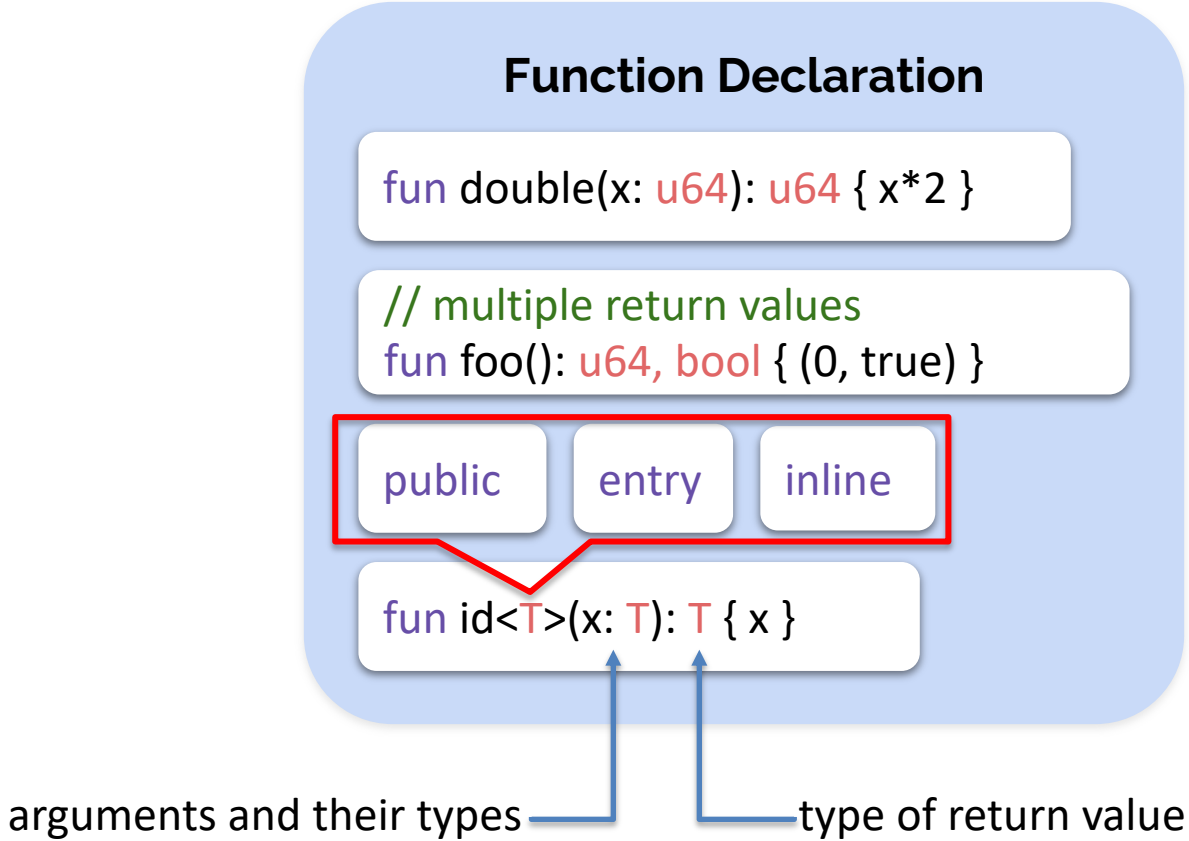
- Signers must match the number of signatures in Tx

```
module my_addr::hello_blockchain {
    use std::string::String;

    struct Message has key, copy, drop {
        message: String
    }

    public entry fun create(
        caller: &signer,        // Tx must contain signature
        msg: String
    ) {
        move_to(caller, Message {message: msg})
    }
}
```

Functions


Function Declaration
fun double(x: u64): u64 { x*2 }
// multiple return values
fun foo(): u64, bool { (0, true) }
public
entry
inline
fun id<T>(x: T): T { x }
arguments and their types
type of return value

Function Signature: Function Body

Here it moves a Message struct to the caller's address

This is the simplest example of global storage

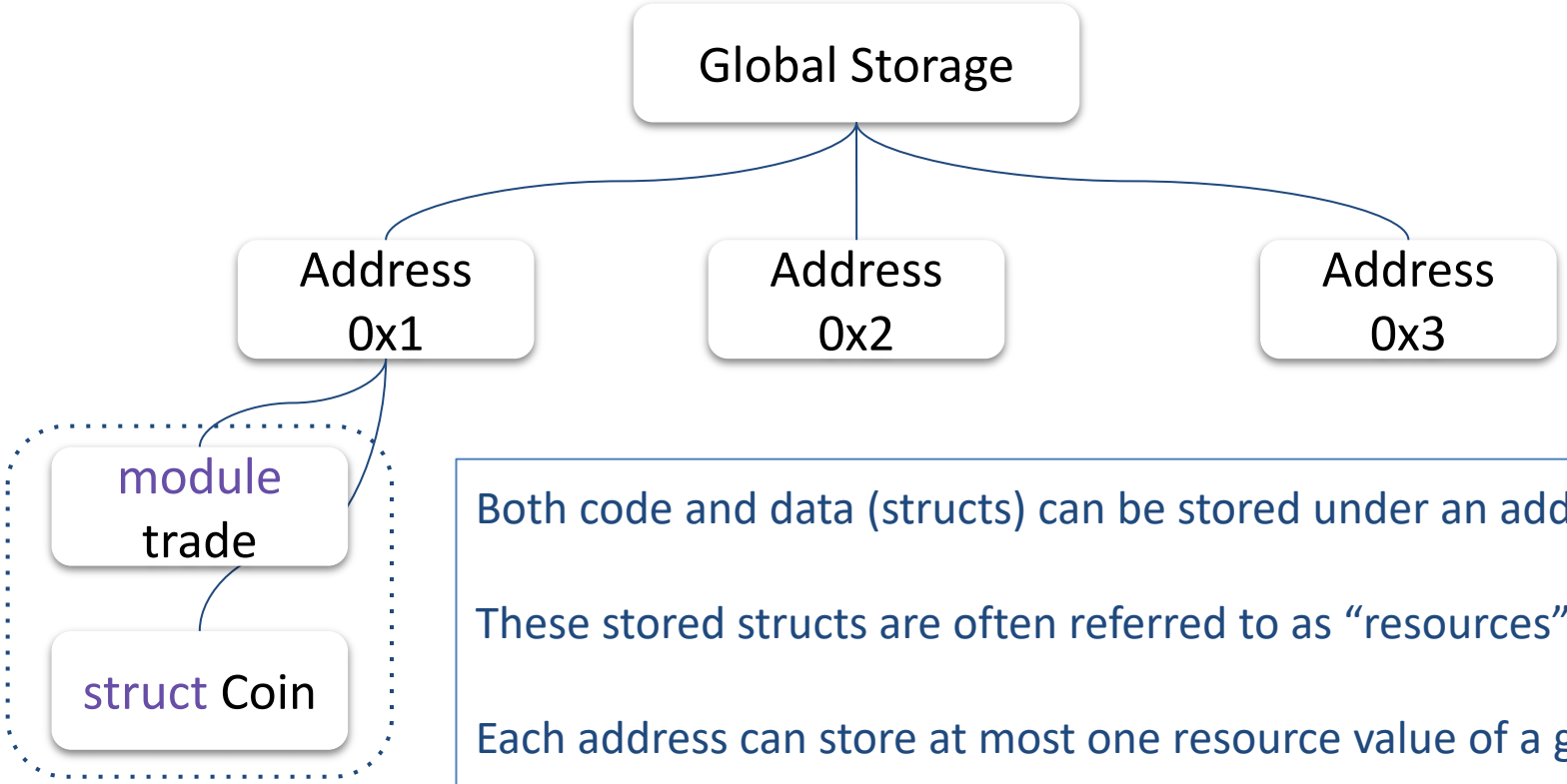
- The Message object is created under address **my_addr** and then moved to **caller**

```
module my_addr::hello_blockchain {
    use std::string::String;

    struct Message has key, copy, drop {
        message: String
    }

    public entry fun create(
        caller: &signer,
        msg: String
    ) {
        move_to(caller, Message {message: msg})
    }
}
```

The Global Storage



Both code and data (structs) can be stored under an address

These stored structs are often referred to as “resources”

Each address can store at most one resource value of a given type

Move offers 4 operations to interact with the global storage

- **exists<T>(addr):** checks whether a resource exists under the given address
- **move_to<T>(&signer, resource)**
  - publishes a new resource under the address of to the given signer
- **move_from<T>(addr):** removes a resource from the given address and returns it
- **borrow_global<T>(addr) / borrow_global_mut<T>(addr)**
  - obtains a reference to a resource under the given address

These ops can only be used within the module that defines the struct

Bonus topic: DAOs

Decentralized Organizations

Decentralized Orgs (DAO)

What is a DAO?

- A Dapp deployed on-chain at a specific address

- Anyone (globally) can send funds to DAO treasury
- Anyone can submit a proposal to DAO
  - ⟹ participants vote
  - ⟹ approved ⇾ proposal executes

snapshot.org

Examples of DAOs

- **Collector DAOs**: PleasrDAO, flamingoDAO, ConstitutionDAO, …

  (see art collection at https://gallery.so/pleasrdao)

PleasrDAO: 103 members.

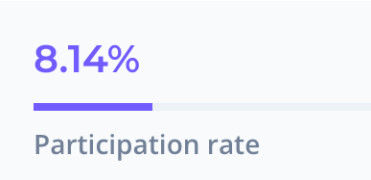
Manages a treasury, has full time employees.

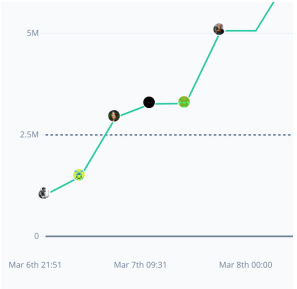
Deliberations over what to acquire over telegram.

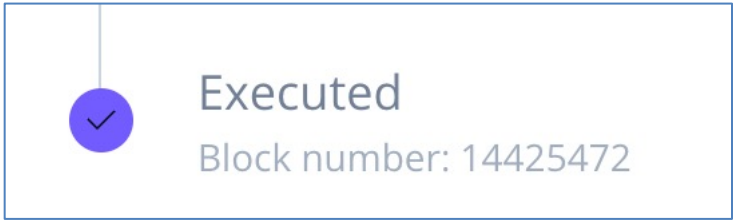
Examples of DAOs

- Collector DAOs: PleasrDAO, flamingoDAO, ConstitutionDAO, …
- **Grants DAOs**: gitcoin (83K members), …

Proposal ID 21: This proposal looks to ratify the allocation of 30,000 GTC from the Community Treasury to the MMM workstream.







(tally.com)

Examples of DAOs

- Collector DAOs: PleasrDAO, flamingoDAO, ConstitutionDAO, …
- Grants DAOs: gitcoin, …
- **Protocol DAOs**: manages operation of a specific protocol
  Uniswap DAO (74K), Compound DAO (8K), …
- **Social DAOs**: FWB, …
- **Investment DAOs**: many

Many DAO governance experiments

Who can vote? How to vote? What voting mechanism?

DAOs: a platform for experimenting with governance mechanisms

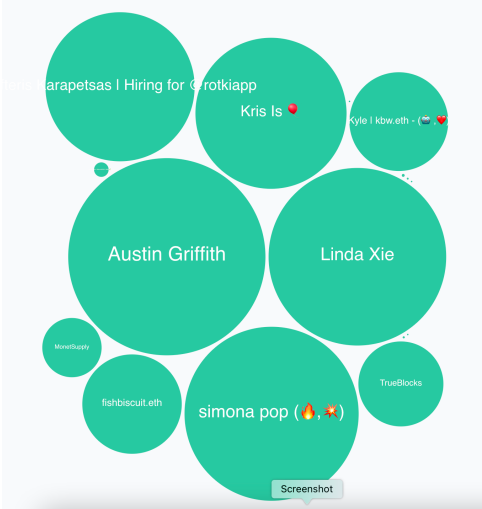
Governance method

One token one vote: (most common)

- Members receive tokens based on their contribution
- Everyone can vote

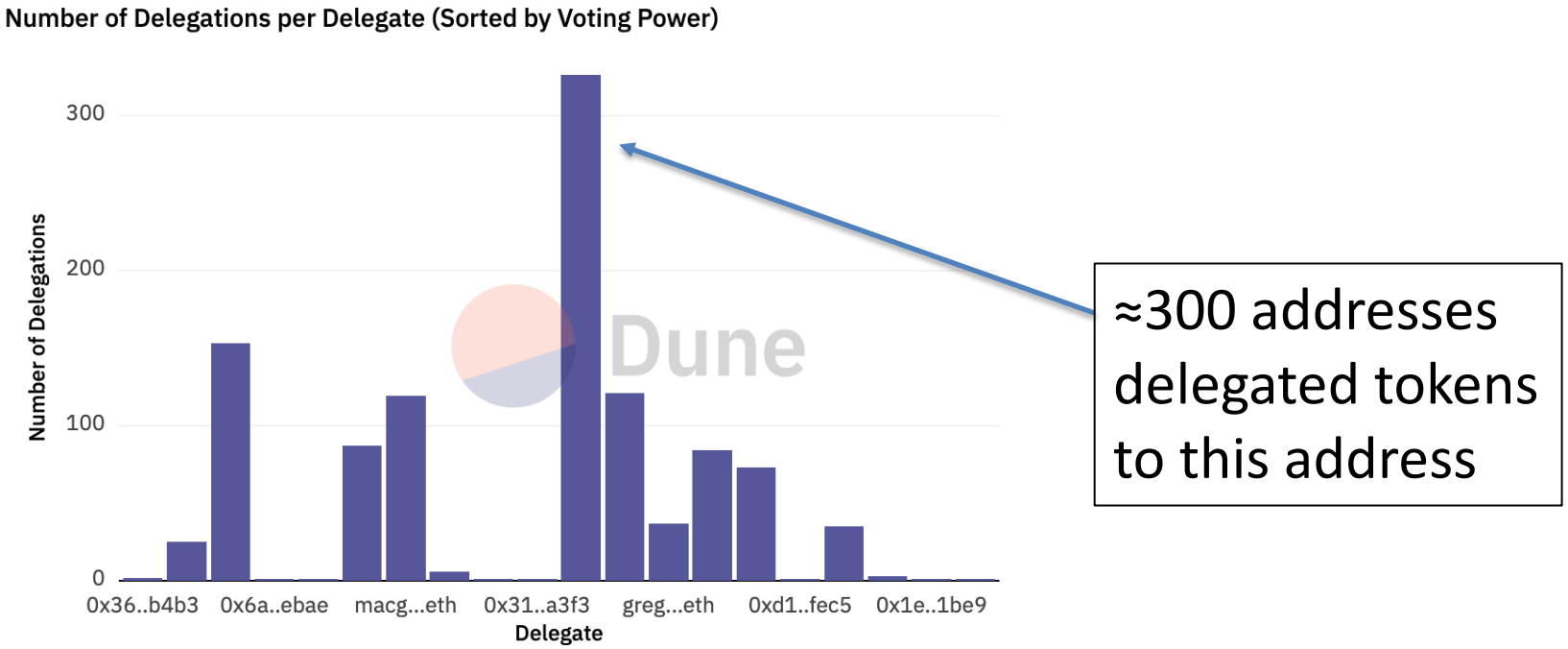
In reality: low participation rate (8%)

⇒ delegation



proposal 21 (tally.com)

Delegation example: element



Governor Bravo

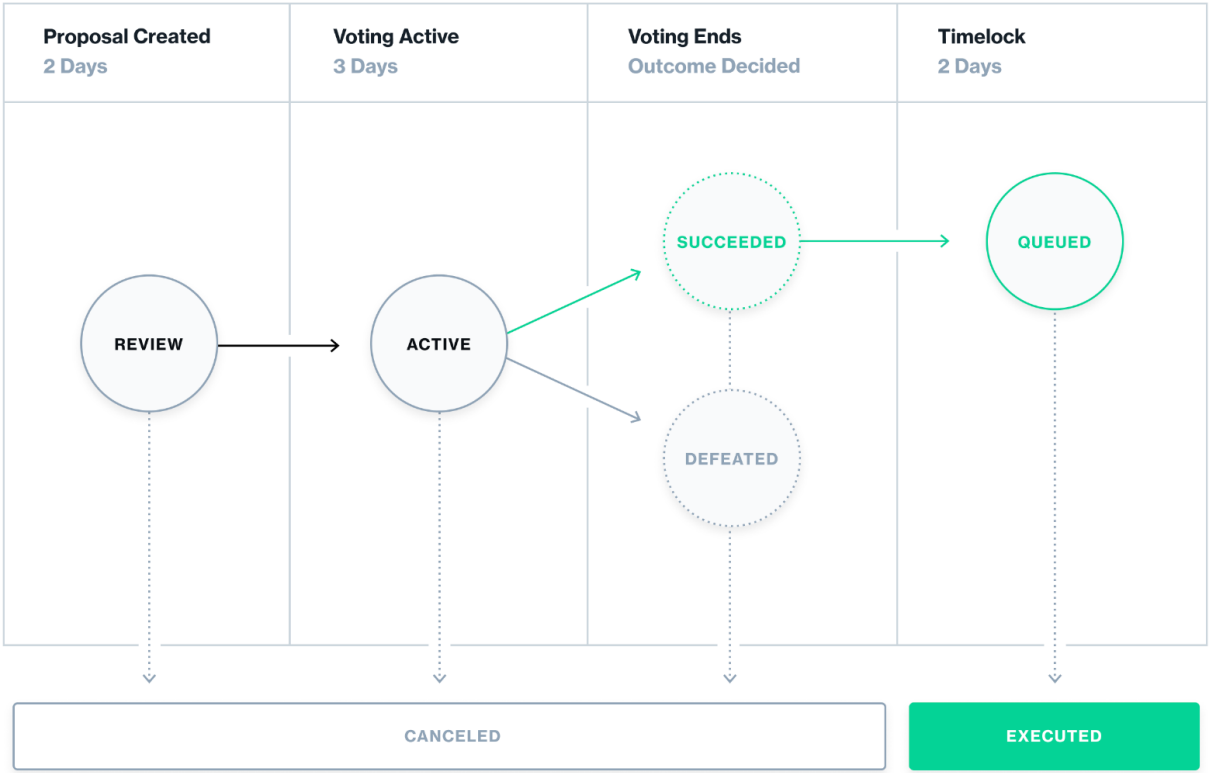
Governor alpha (Jan. 2020) updated to **Bravo** (March 2021):

- A collection of smart contracts used for governance
- Forked and used by many other protocols

Other alternatives:

- OpenZeppelin Governor protocol

The life cycle of a proposal



Governor Bravo

Implements an ERC-20 governance token. Let's call it GOV.

- An address that owns GOV tokens can vote on proposals

The proposal I am voting for
(with all my allotted voting tokens)

0: against
1: in favor
2: abstain

```
function castVote(uint proposalID, uint8 support)
```

(cannot vote partially)

Governor Bravo

Implements an ERC-20 governance token. Let's call it GOV.

… or delegate its GOV tokens to another address

function **delegate**(address delegatee)

delegate all my voting tokens

can re-delegate at any time by calling function again

An address can only delegate all its tokens to one address at a time

… but I can hold my GOV tokens at multiple addresses

Creating a proposal

```
function propose(
        address[] targets,
        uint[]    values,
        string[]  signatures,
        bytes[]   calldatas,

        string description)      //  proposal description
                                        (human readable)
    returns (uint)
```

The function to call on governor contract

If proposal passes,
how to execute it:
what functions to call (actions)
and with what arguments

The ID of newly created proposal

Global contract parameters

- **quorumVotes**: the minimum # of votes for a proposal to pass
- **proposalThreshold**: min # votes needed to create a proposal
- **votingDelay**: # blocks to wait until voting can begin after a proposal is created (e.g., two days)
- **votingPeriod**: # blocks when voting is open (e.g., three days)

All these parameters can be changed by governance in Bravo

Proposal execution

Once proposal gets enough votes and delays expire:

- anyone can call: function **execute**(uint proposalID)

⇒ prescribed functions in proposal get called

The limits of democracy

Anytime before execution

- “guardian” can call **cancel()** to cancel a proposal

Who is this guardian??

- set at Governor contract creation time
- can abdicate at any time by calling **_abdicate**()

DAO privacy questions

- **Private DAO participation**: keep membership list private
- **Private voting**: keep who voted how on each proposal private
- **Privately delegate** voting rights
- **Private treasury**

… while complying with all relevant laws.

Some DAOs made mistakes and were attacked

- Attack on Beanstalk governance: $182M
- Attack on Yam Finance governance: thwarted
- Attack on Build Finance governance $1.5M
- Attack on Mango governance: $115M
- Steem governance drama

What happened? What can we learn?

(1) Beanstalk (Apr. 2022)

An Ethereum-based stablecoin **Bean**. Governance token **Stalk.**

The problem:

- an attacker can buy Stalk tokens,
- submit a proposal,
- vote on proposal, and
- have proposal execute

all in a single transaction

Beanstalk: the attack

Recall: **flashloan** (has many applications)

- a loan that is taken out and repaid back in a single transaction
- No risk to lender ⇒ unbounded amount can be borrowed

The attack:

- Attacker took a huge flashloan from Aave, bought lots of Stalk,
- Passed a proposal to pay $80M to the attacker from the treasury,
- Sold the Stalk, and repaid the flashloan,
- Sent the proceeds to Tornado cash (donated $250K to Ukraine)

The lesson

The protocol relaunched four months later.

The lesson: governance requires delays

- Require token holders to hold token for a long period of time
- Build a delay between proposal genesis, voting, and execution

(2) Yam Finance (July 2022)

- A DeFi project running on Ethereum. Governed token **YAM**.

What happened? (within a single day)

- Attacker bought 224739 YAM with borrowed funds
- Attacker submitted a governance proposal
  granting it control of project reserves ($3M USD)
- Voted on proposal with borrowed 224739 YAM, hitting quorum
- Borrowed YAM exchanged for Eth and paid back

What happened next?

Yam Finance team: noticed the proposal shortly after it hit quorum

⇒ retained **cancel** power, and canceled the proposal.

Lesson:

- Illustrates the positive power of a veto
- The problem: power can never be taken away
  (unless voluntarily given away)

END OF LECTURE

Next lecture: Privacy on chain