Blockchains & Cryptocurrencies

Ethereum Applications / DeFi

Instructor: Matthew Green
Fall 2024

News?

Review: How to upgrade a contract?

How to upgrade a contract?

- Contracts are default immutable
- If there is no useful ongoing state, don't: just replace it
- If there is ongoing state (e.g., account balances) then:
  - Don't allow upgrades — and pray you got the code right
  - Call upgradeable/replaceable library code
  - Create a complex mechanism to transfer state from an old contract instance to a new contract instance

How to upgrade a contract?

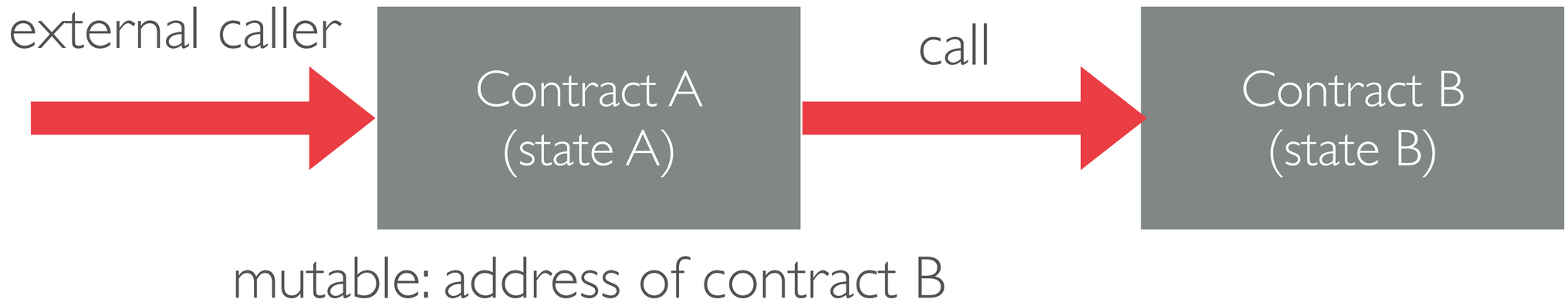
- Contracts are default immutable
- If there is no useful ongoing state, don't: just replace it
- If there is ongoing state (e.g., account balances) then:
  - Don't allow upgrades — and pray you got the code right
  - Call upgradeable/replaceable library code
  - Create a complex mechanism to transfer state from an old contract instance to a new contract instance

Proxy pattern

- Based on two standard features of Ethereum
  - An Ethereum contract (A) can call another contract (B) as a library

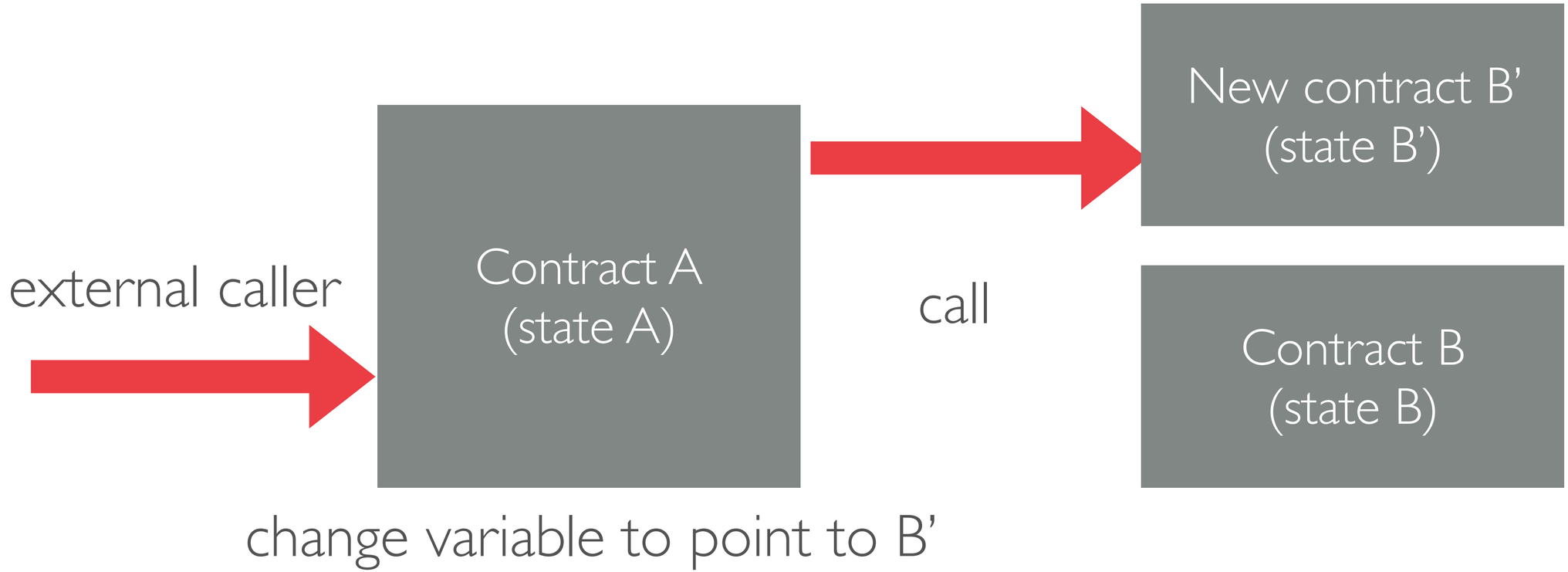
Proxy pattern

- Based on two standard features of Ethereum
  - An Ethereum contract (A) can call another contract (B) as a library
  - The address of that second contract can be stored in a mutable variable



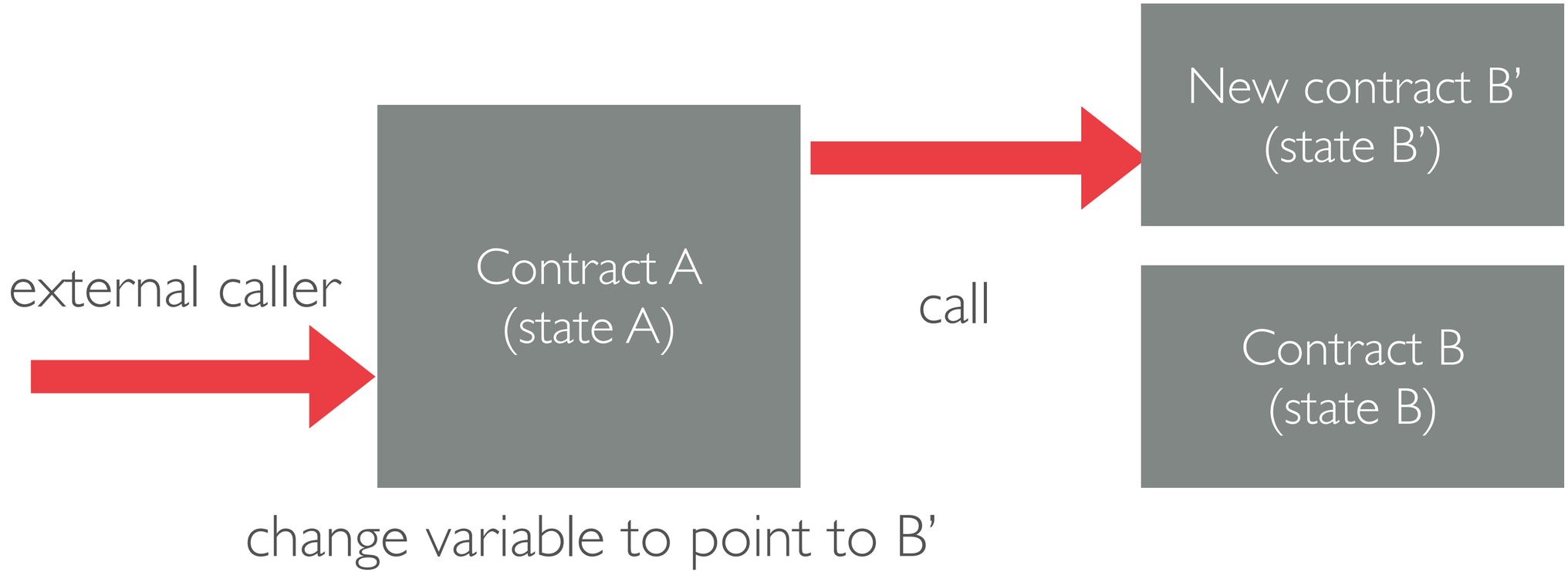
Proxy pattern

- Based on two standard features of Ethereum
  - We can always update that variable (using some special contract call we implement ourselves.)



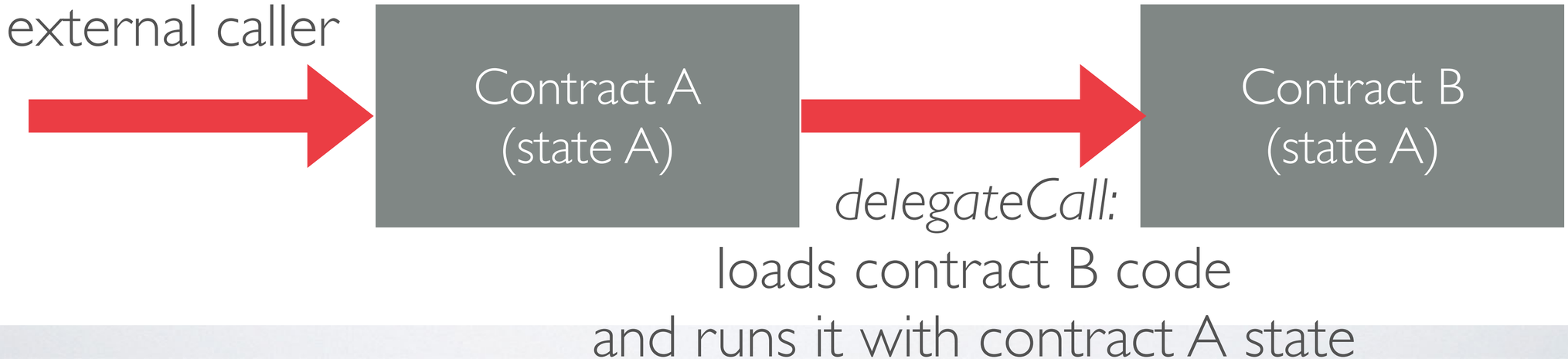
Proxy pattern

- **What are some problems with this solution (so far)?**



Delegate-Call

- Ethereum offers a nice "optimization"
  - A contract A can "load the code" from contract B into its own state context, run code B on A's state
  - This is called delegateCall
    - This is an exception to our normal isolation rules



```
assembly {
  // (1) copy incoming call data
  calldatacopy(0, 0, calldatasize())

  // (2) forward call to logic contract
  let result := delegatecall(gas(), implementation, 0, calldatasize(), 0, 0)

  // (3) retrieve return data
  returndatacopy(0, 0, returndatasize())

  // (4) forward return data back to caller
  switch result
  case 0 {
      revert(0, returndatasize())
  }
  default {
      return(0, returndatasize())
  }
}
```

Proxies: advantages/disadvantages

- The ability to upgrade contract code is useful (bug fixes, etc.)
  - Counterargument: it can be very risky
  - If someone "hacks" your upgrade mechanism, they can steal all your money
  - What are some ways we can secure this upgrade process?
    - **Multi-sig** (require many signatures to upgrade contract)
    - **Voting** (require many token-holders to vote for an upgrade)
    - **Timelocks** (put a delay between start & end of an upgrade)

Concurrency and re-entrancy

- Ethereum transactions run sequentially and atomically
  - In principle this is good: there **appear** to be no concurrency issues (no threads) and your methods always run to completion or don't complete at all!

Example (ok)

```
function transfer(uint amount, address recipient) … {
    if (balances[msg.sender] < amount) {
        // insufficient balance
        revert('Something bad happened');
    }

    // Transfer the money
    balances[msg.sender] -= amount;
    balances[recipient] += amount;
}
```

Example (problematic)

```
function transferAndNotify(uint amount, address notifyContract) … {
    if (balances[msg.sender] < amount) {
        // insufficient balance
        revert('Something bad happened');
    }

    // Call the specified contract to notify it that a deposit is coming
    notifyContract.notify(amount, "You are getting a deposit!");

    // Transfer the money
    balances[msg.sender] -= amount;
    balances[notifyContract] += amount;
}
```

Re-entrancy

- There is still one scary "gotcha"!
- Ethereum is not re-entrancy "safe". *You can still have multiple calls to the same routine within any given call-stack.*

Re-entrancy

```
function transferAndNotify(uint amount, address notifyContract) … {
    if (balances[msg.sender] < amount) {
        // insufficient balance
        revert('Something bad happened');
    }
    balances[notifyContract] += amount;

    // Call the specified contract to notify it that a deposit is coming
    notifyContract.notify(amount, "You are getting a deposit!");

    // Transfer the money
    balances[msg.sender] -= amount;
}
```

Re-entrancy

```
function transferAndNotify(uint amount, address notifyContract) … {
    if (balances[msg.sender] < amount) {
        // insufficient balance
        revert('Something bad happened');
    }
    balances[notifyContract] += amount;

    // Call the specified contract to notify it that a deposit is coming
    notifyContract.notify(amount, "You are getting a deposit!");

    // Transfer the money
    balances[msg.sender] -= amount;
}
```

What if this contract call calls us?

Re-entrancy

```
function transferAndNotify(uint amount, address notifyContract) … {
    if (balances[msg.sender] < amount) {
        // insufficient balance
        revert('Something bad happened');
    }
    balances[notifyContract] += amount;

    // Call the specified contract to notify it that a deposit is coming
    notifyContract.notify(amount, "You are getting a deposit!");

    // Transfer the money
    balances[msg.sender] -= amount;
}
```

What if this contract call calls us?

Re-entrancy

```
function transferAndNotify(uint amount, address notifyContract) … {
    if (balances[msg.sender] < amount) {
        // insufficient balance
        revert('Something bad happened');
    }
    balances[notifyContract] += amount;

    // Call the specified contract to notify it that a deposit is coming
    notifyContract.notify(amount, "You are getting a deposit!");

    // Transfer the money
    balances[msg.sender] -= amount;
}
```

What if this contract call calls us?

Re-entrancy

```
function transferAndNotify(uint amount, address notifyContract) … {
    if (balances[msg.sender] < amount) {
        // insufficient balance
        revert('Something bad happened');
    }
    balances[notifyContract] += amount;

    // Call the specified contract to notify it that a deposit is coming
    notifyContract.notify(amount, "You are getting a deposit!");

    // Transfer the money
    balances[msg.sender] -= amount;
}
```

What if this contract call calls us?

DAO disaster (2016)

- Decentralized Autonomous Organization
  - "Like a VC fund" but decentralized
  - Implementation: a contract that controls money, and directs its disbursement according to "shareholder votes"
  - Shareholders buy in, pool their ETH (sending to contract)
  - Then vote on investments, which are made together
  - Users can "split" a DAO

“The DAO”

```
function splitDAO(
  uint _proposalID,
  address _newCurator
) noEther onlyTokenholders returns (bool _success) {

  ...
  // XXXXX Move ether and assign new Tokens.  Notice how this is done first!
  uint fundsToBeMoved =
      (balances[msg.sender] * p.splitData[0].splitBalance) /
      p.splitData[0].totalSupply;
  if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender)
== false) // XXXXX This is the line the attacker wants to run more than once
      throw;

  ...
  // Burn DAO Tokens
  Transfer(msg.sender, 0, balances[msg.sender]);
  withdrawRewardFor(msg.sender); // be nice, and get his rewards
  // XXXXX Notice the preceding line is critically before the next few
  totalSupply -= balances[msg.sender]; // XXXXX AND THIS IS DONE LAST
  balances[msg.sender] = 0; // XXXXX AND THIS IS DONE LAST TOO
  paidOut[msg.sender] = 0;
  return true;
}
```

Solutions?

```
function transferAndNotify(uint amount, address notifyContract) … {
    if (globalLock == true) {
        revert("Locked.");
    }
    globalLock = true;

    if (balances[msg.sender] < amount) {
        // insufficient balance
        revert('Something bad happened');
    }

    // Call the specified contract to notify it that a deposit is coming
    notifyContract.notify(amount, "You are getting a deposit!");

    // Transfer the money
    balances[msg.sender] -= amount;
    balances[notifyContract] += amount;

    globalLock=false;
}
```

```
function transferAndNotify(uint amount, address notifyContract) … {
   if (globalLock == true) {
      revert(“Locked.”);
   }
   globalLock =

   if (balances[msg.s
      // insufficient b
      revert(‘Someth
   }

   // Call the specifi
   notifyContract.no

   // Transfer the m
   balances[msg.sen
   balances[notifyContract] += amount;

   globalLock=false;
}
```

Drawbacks of (global) locks:

1. Extra gas (due to stores/loads)

2. Can get “stuck” if you’re careless

3. Sometimes re-entrant calls are useful!

Check-Effects-Interaction pattern

- Most common solution is to follow a code pattern:
  - First perform all contract checks (CHECKS)
  - Second, update contract state (EFFECTS)
  - Finally, make any contract calls (INTERACTION)

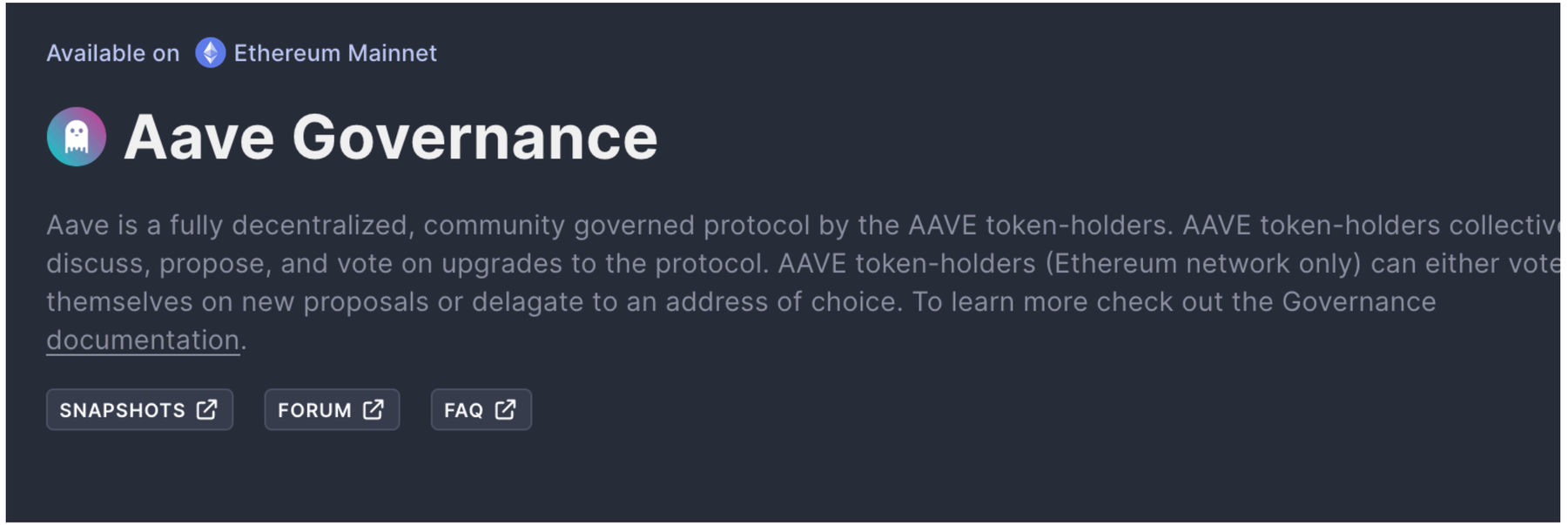
Check-Effects-Interaction pattern

```
function transferAndNotify(uint amount, address notifyContract) … {
   // CHECK
   require (amount < balances[msg.sender]);

   // EFFECTS
   balances[msg.sender] -= amount;
   balances[notifyContract] += amount;

   // INTERACTION
   notifyContract.notify(amount, "You are getting a deposit!");
}
```

Contract governance

- The ability to upgrade contracts can be even more powerful
  - Enables the notion of "voting"-based contract governance
- **Idea**: issue a "governance token" (e.g., ERC20)
  - People might get the token in exchange for depositing real money, or for other reasons
  - This token gives holders a "vote" on contract governance
  - People can propose new features for the contract (in practice, this is essentially a contract upgrade proposal)
  - Voters can then review/vote/adopt the new features

Contract governance



Contract governance: dark side

- An observation in practice:
  - Contract governance is sometimes a form of "regulatory arbitrage"
  - Often: the contract developers want to argue that they are simply a software developer, and do not control this "decentralized protocol"
  - However they (or their friends/partners/VCs) may in fact control the bulk of the voting tokens
    - Hence one should always be a little bit skeptical of these claims

Contract governance: dark side II

News ▸ Technology

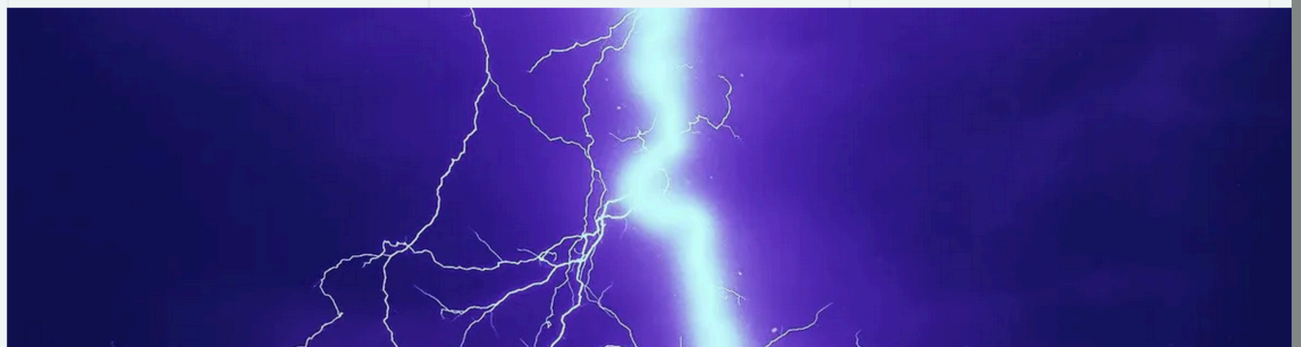
Group Uses Flash Loan to Game Maker Protocol Governance

BProtocol used a DeFi arbitrage tool to push through a governance vote it had proposed on the Maker protocol.

By Jeff Benson

Oct 29, 2020

2 min read



I intend to return to this (if there is time)…

Standards Track: ERC

ERC-2535: Diamonds, Multi-Facet Proxy

Create modular smart contract systems that can be extended after deployment.

Authors Nick Mudge (@mudgen)

Created 2020-02-22

Table of Contents

As state space

Decentralized Exchanges

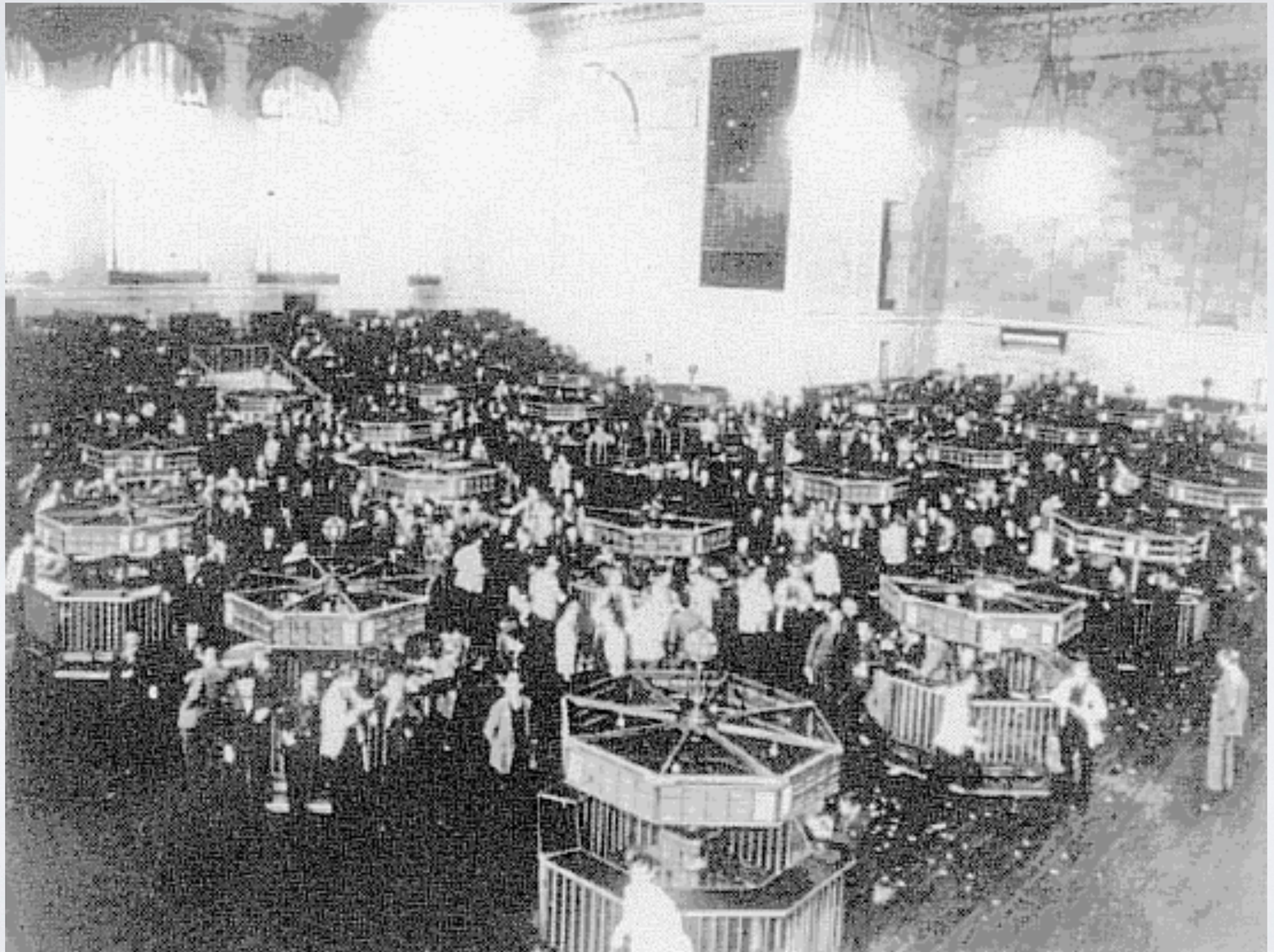
- Given multiple assets (ERC20s) on Ethereum, can we build platforms to exchange them?
  - Breaks down into two-subproblems:

    1. Price discovery / order matching
    2. Execution (swap)

This part is relatively easy

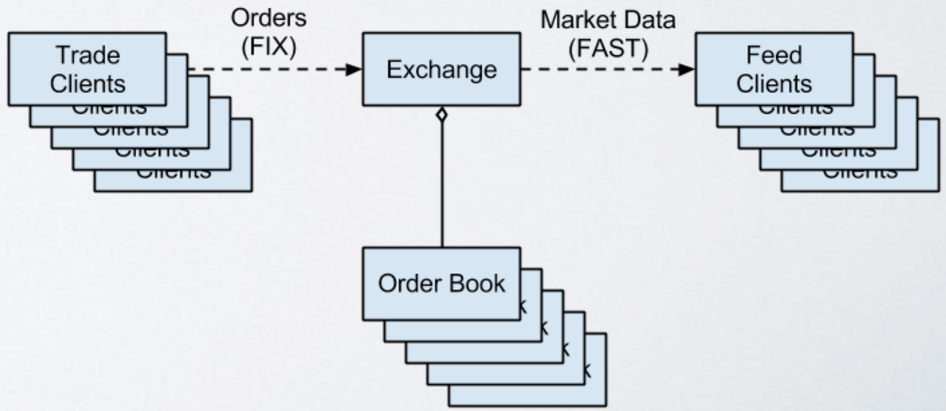
DeFi!

Traditional (centralized) exchange

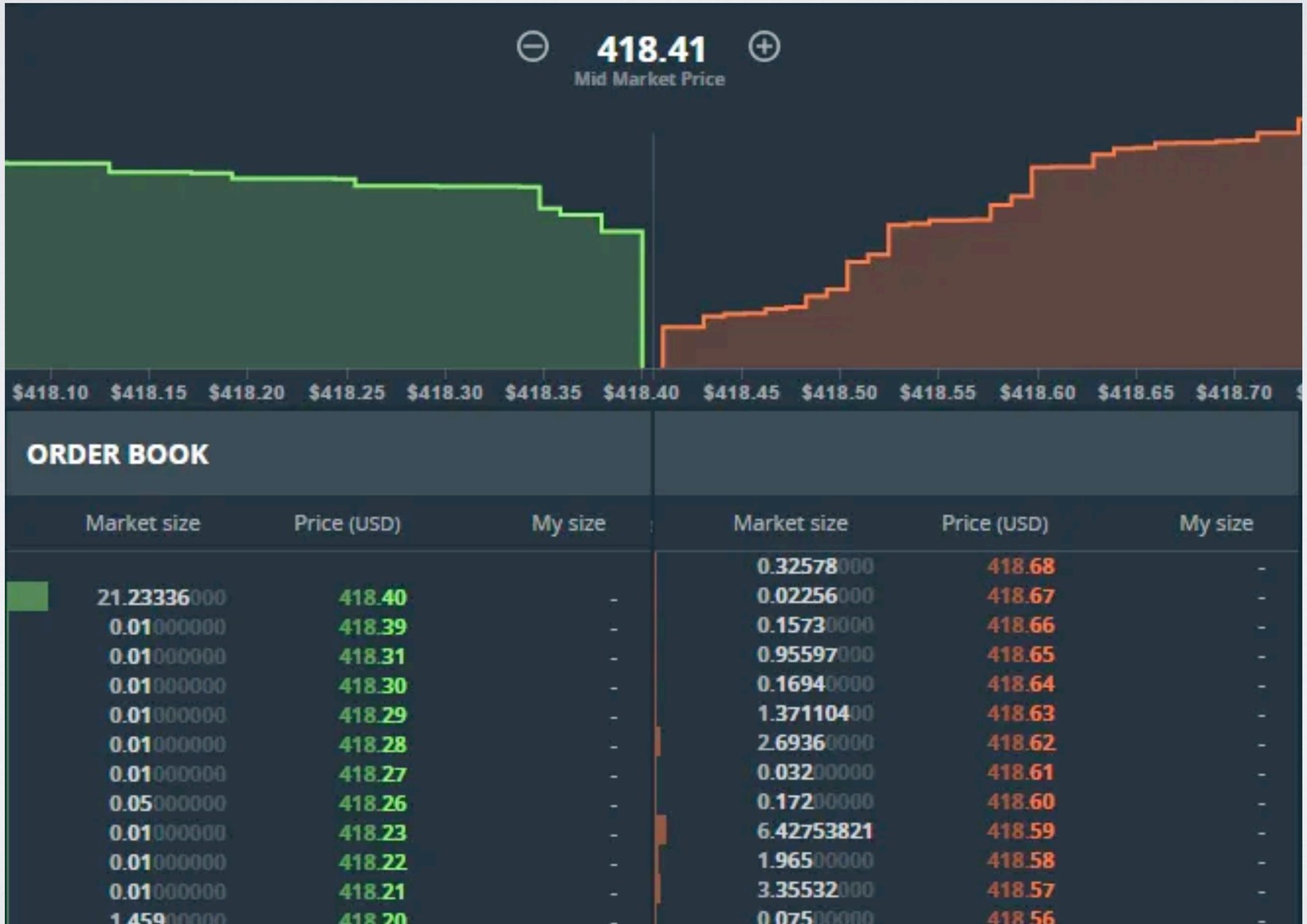


Electronic centralized exchange

- Maintains an "electronic order book"
  - Receives order of various types (e.g., market, limit, etc.)
  - Implements a matching engine to reconcile orders
  - Executes trades (swaps)
  - Produces asset price data (as a side effect)



Electronic order book

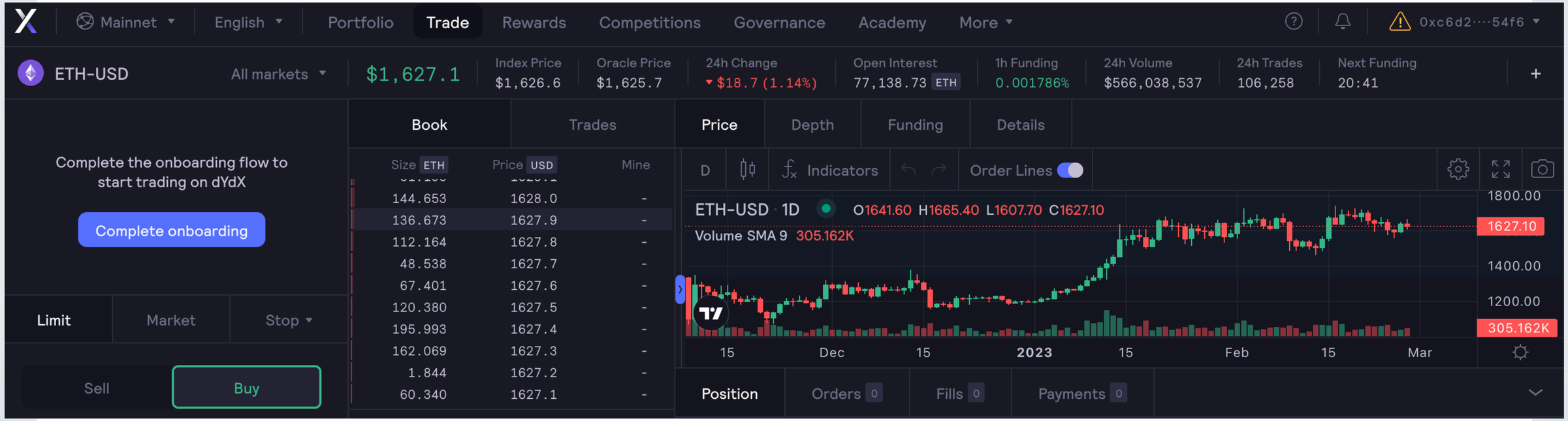


Electronic order book

- Can we do this purely on Ethereum?
  - Why/why not?

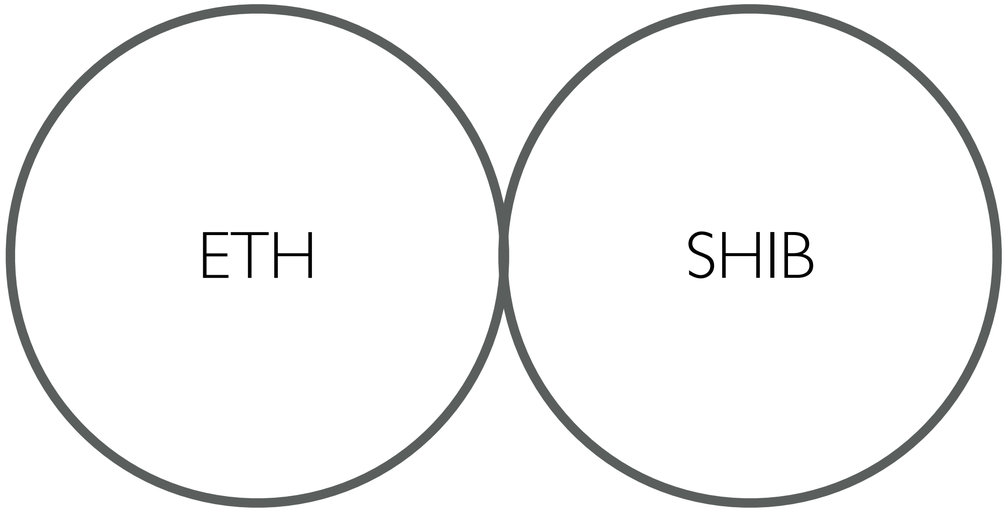
Electronic order book

- Can we do this on Ethereum?
  - Why/why not?



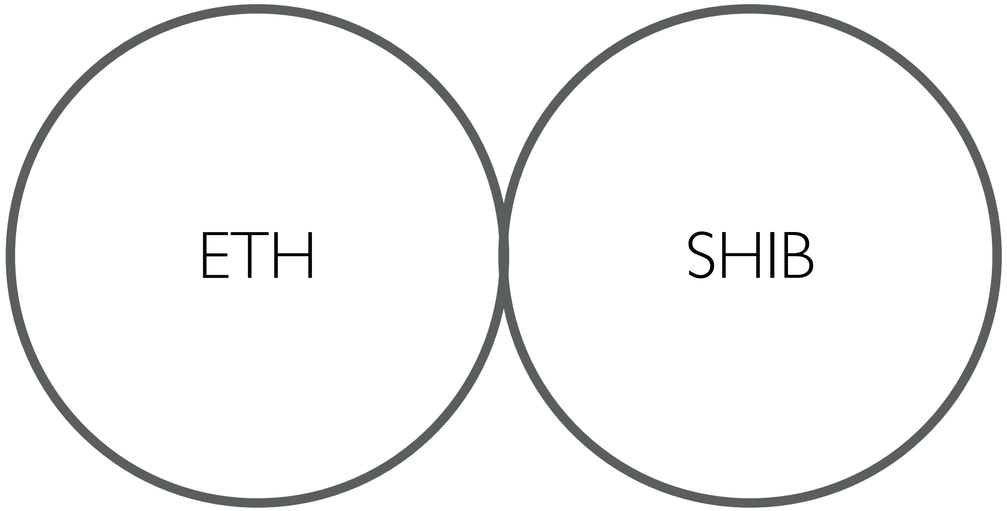
Constant Function Market Makers (AMM)

- A simpler (and much less efficient) type of exchange
  - Built from pairs of assets (e.g., ETH/SHIB)
  - Liquidity providers can deposit and withdraw asset pairs (in some ratio), forming a "liquidity pool"



Constant Function Market Makers (AMM)

- A simpler (and much less efficient) type of exchange
  - Built from pairs of assets (e.g., ETH/SHIB)
  - New LPs can increase/reduce the overall size of both pools, without changing the ratio

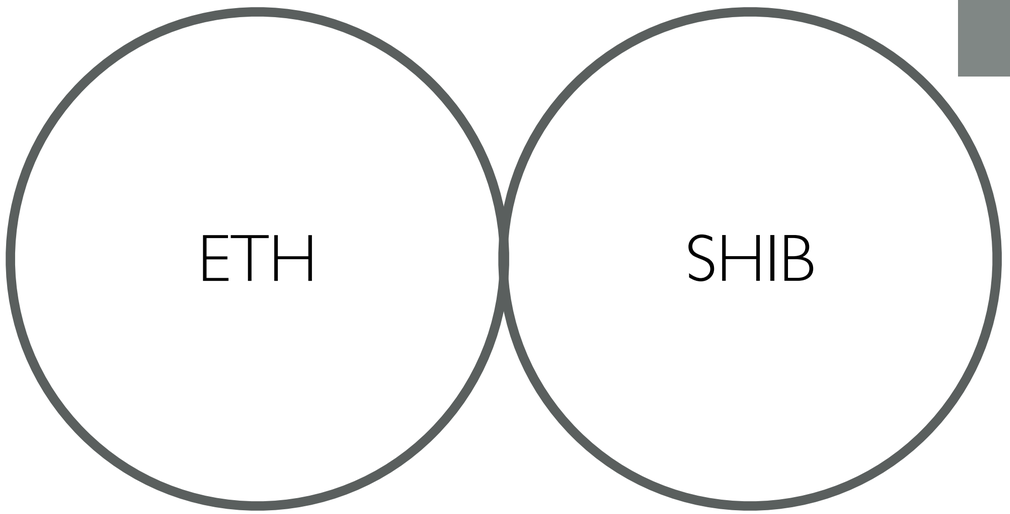


Constant Function Market Makers (AMM)

- A simpler (and much less efficient) type of exchange
  - Built from pairs of assets (e.g., ETH/SHIB)
  - Users can also "swap" (deposit one asset, withdraw the other), in exchange for fees

Overall goal:
Preserve a constant function (e.g., product, sum). E.g.,:

$$A * B = K$$

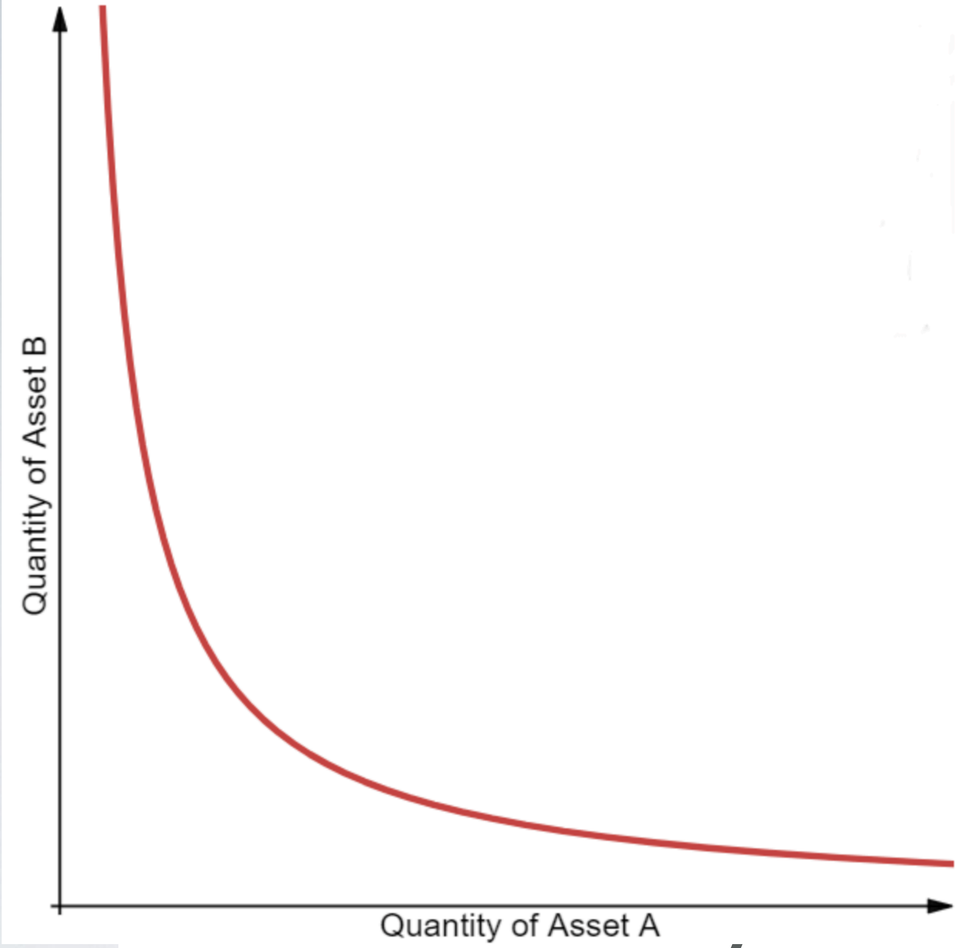


ɔn Market Makers (AMM)

ess efficient) type of exchange

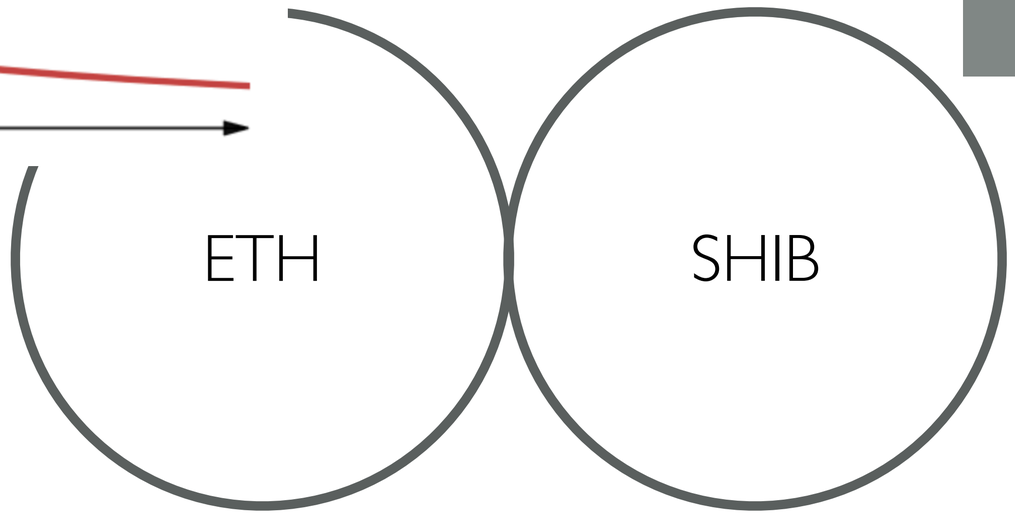
sets (e.g., ETH/SHIB)

d withdraw assets (in vari
ɔol”



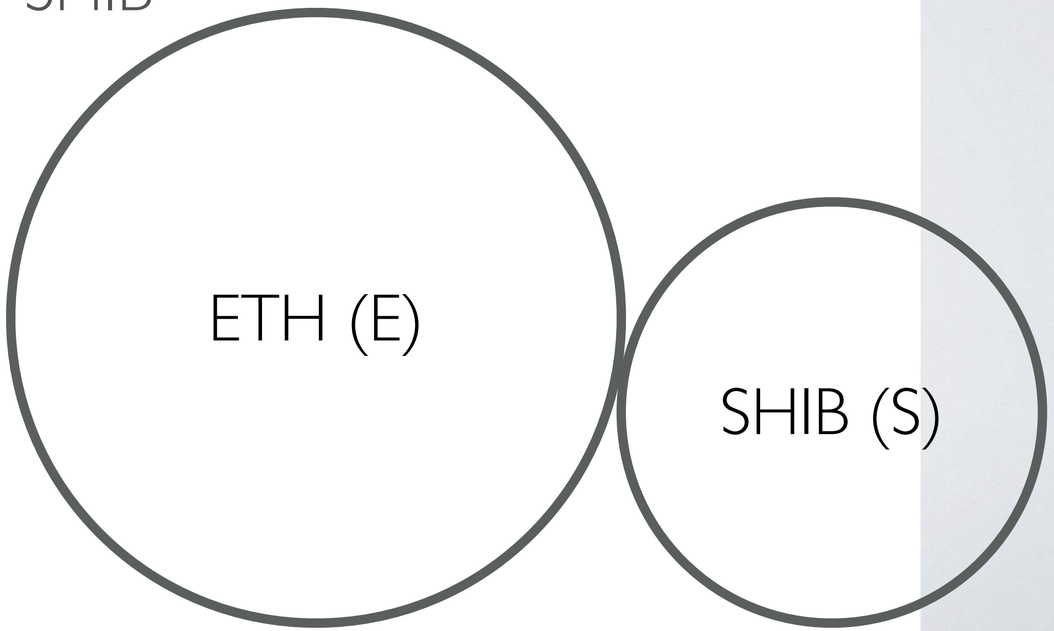
Overall goal:
Preserve a constant function (e.g., product, sum). E.g.,:

$A * B = K$

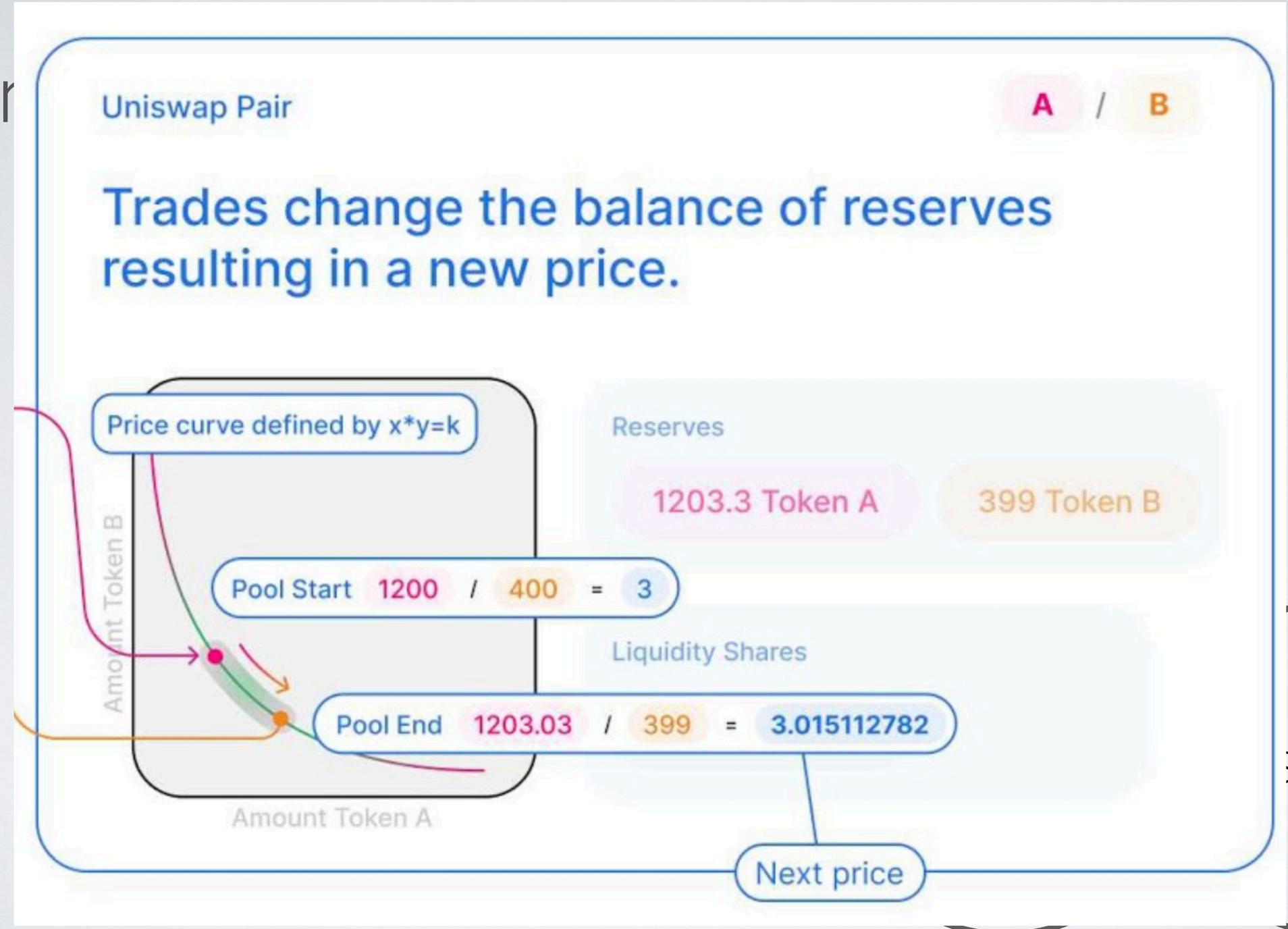


Constant Function Market Makers (AMM)

- User A deposits ETH, wants to “buy" SHIB
  - Initial equation: E * S = K
  - Defines an implicit price (ratio) between the assets
  - They can now deposit ETH & withdraw SHIB at that price (plus pay some fees)
    - This trade changes the market price
    - New price ratio for SHIB: (K / E’) / S’



Cor M)

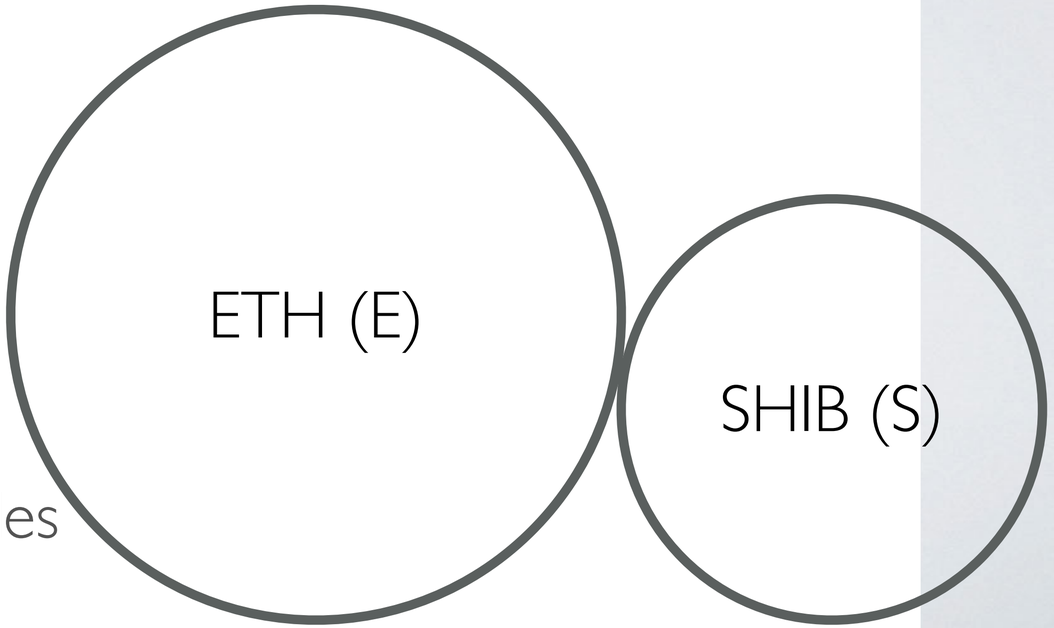


SDC (U)

https://academy.shrimpy.io/lesson/what-is-an-amm-automated-market-maker

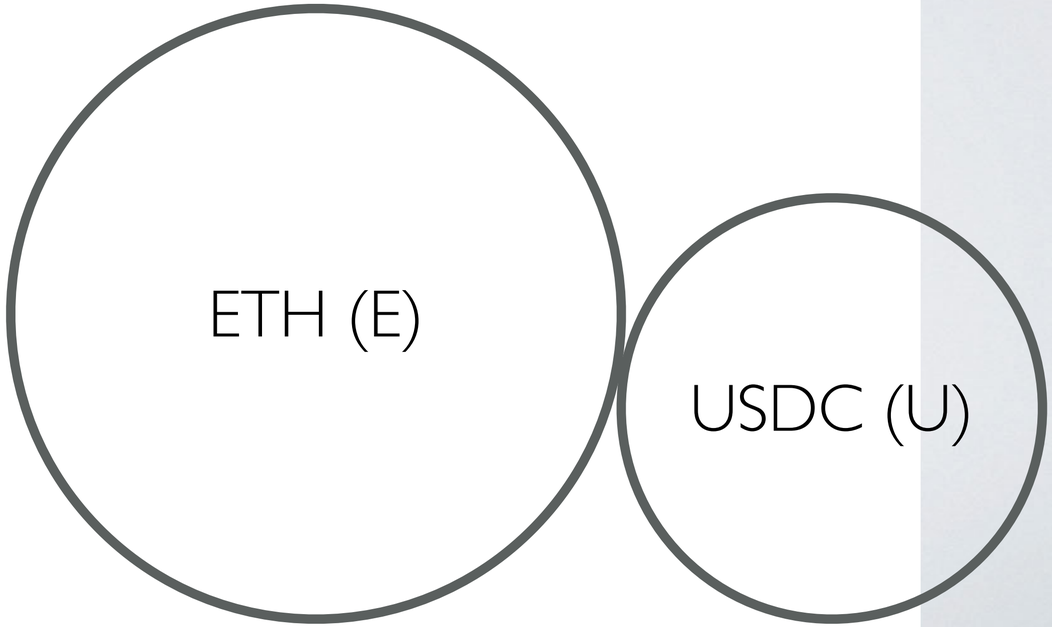
Arbitrage / MEV

- In principle, the price of assets should stay "close" to the true price (whatever that is)
  - If the ratio gets far out of whack, hungry traders will "adjust" the pool sizes by executing profitable swaps
  - If there is no pool for an asset pair, systems can "route" trades through multiple pools with more liquidity
    e.g., ETH->USDC, USDC->SHIB
  - Slippage can be quite large
  - Relatively easy to "front-run" public trades

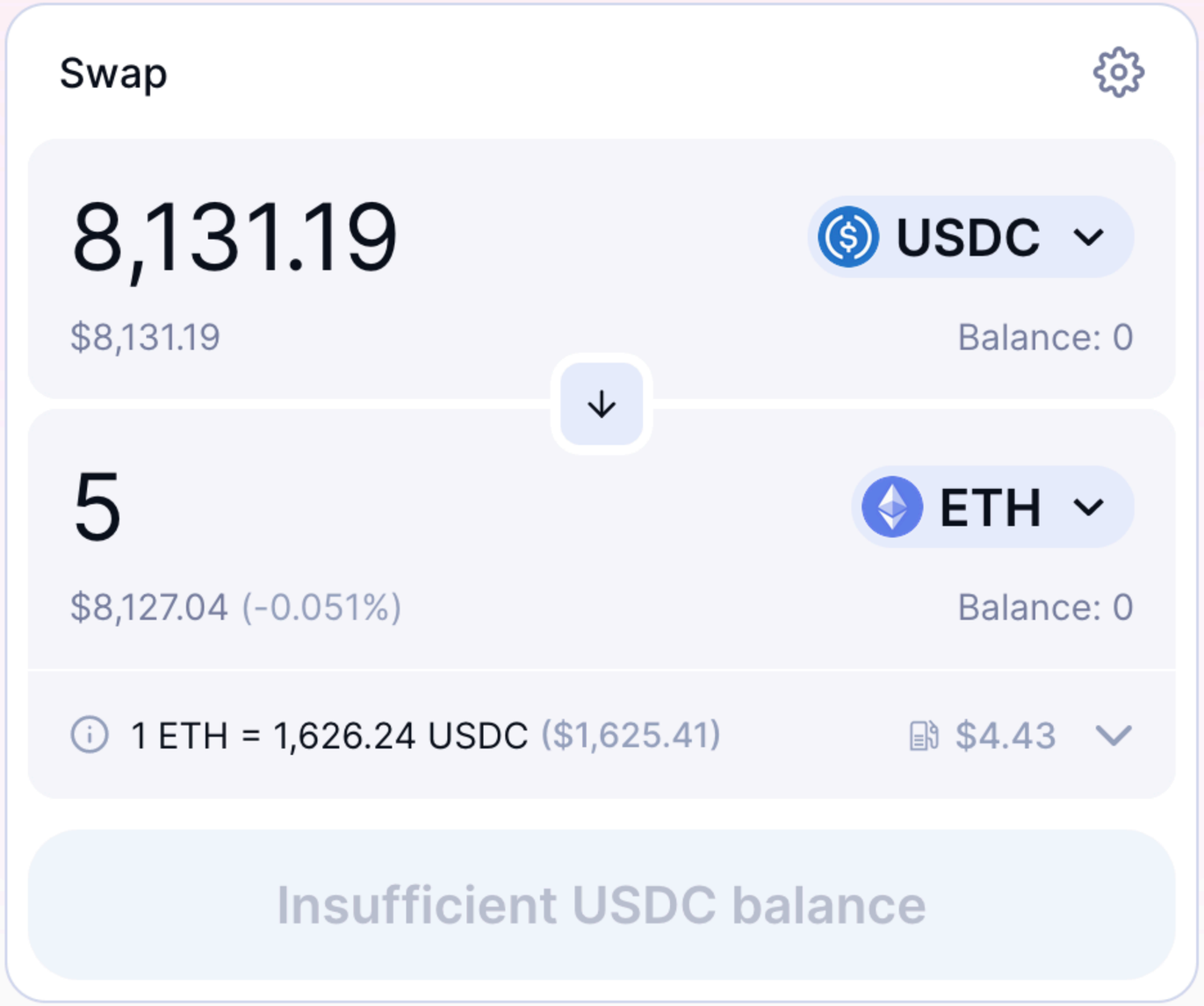


Why put your "liquidity" in these pools?

- The liquidity providers receive fees from the trade (usually split with the DeFi operators, developers)
  - They receive "Liquidity Provider Tokens" (yet another made-up ERC20) that they can later redeem to get money back out (in a new ratio)
  - **Criticism:** being an LP is dumb, because arbitrage is profitable and you are the loser in all those trades
  - Many LPs have been "incentivized" to participate by weird token schemes



W s?

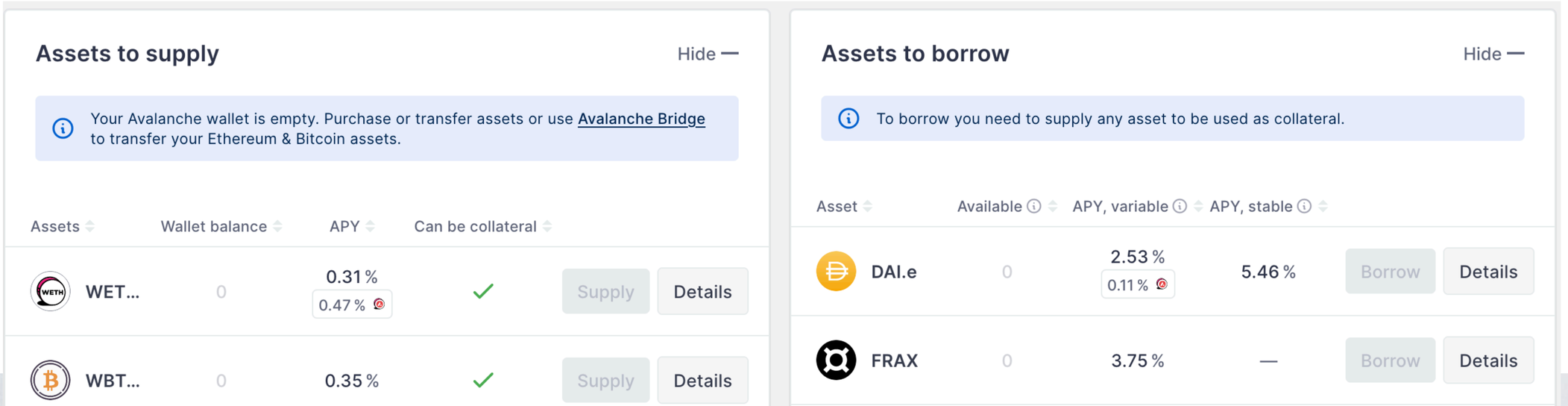

Swap
8,131.19
USDC
$8,131.19
Balance: 0
5
ETH
$8,127.04 (-0.051%)
Balance: 0
1 ETH = 1,626.24 USDC ($1,625.41)
$4.43
Insufficient USDC balance

Asset lending

- Another thing smart contracts can do easily is asset lending
  - **Problem:** smart contracts are not great at getting people to repay asset loans
  - This means you need some way to insure those loans
  - Answer: require (over)collateralization

Asset lending

- Another "popular" DeFi service is collaterized asset lending (borrowing)
  - You wish to borrow some amount of Asset B (for whatever)
  - You can deposit and "lock up" some amount of Asset A as collateral for the loan (and pay interest)



Asset lending

- If lenders don't repay the principal, their collateral is automatically liquidated to repay the lenders
  - Hence collateral value must (substantially) exceed loan value
  - If the value of the collateral drops, the system may automatically liquidate it without warning
  - **Question:** How do these smart contracts know the "value" of anything?

Asset lending

- If lenders don't repay the principal, their collateral is automatically liquidated to repay the lenders
  - Hence collateral value must (substantially) exceed loan value
  - If the value of the collateral drops, the system may automatically liquidate it without warning
  - **Question:** How do these smart contracts know the "value" of anything?
  - **Answer:** services provide price "oracles" to these systems, by sending Txes (to some contract) that contain current prices, e.g., ChainLink.
    (Or they can query AMM contracts!)

Asset lending

Assets to supply

Hide —

Your Avalanche wallet is empty. Purchase or transfer assets or use Avalanche Bridge to transfer your Ethereum & Bitcoin assets.

Assets	Wallet balance	APY	Can be collateral		
WET...	0	0.31 % 0.47 %	✓	Supply	Details
WBT...	0	0.35 %	✓	Supply	Details
WAVAX	0	1.65 % 1.15 %	✓	Supply	Details
AVAX	0	1.65 % 1.15 %	✓	Supply	Details

Assets to borrow

Hide —

To borrow you need to supply any asset to be used as collateral.

Asset	Available	APY, variable	APY, stable		
DAI.e	0	2.53 % 0.11 %	5.46 %	Borrow	Details
FRAX	0	3.75 %	—	Borrow	Details
MAI	0	3.51 %	—	Borrow	Details
USDC	0	2.31 % 0.17 %	5.43 %	Borrow	Details

Asset lending (horror stories)

- There are big risks here, if the collateral asset is badly priced (or thinly traded)
  - Show up with some fake nonsense-coin, that has been "wash traded" into appearing to have value
  - Borrow actual money with it
  - Walk away and never come back
- So these systems are not usually unsupervised… and lenders can lose money

Flash loans!

- It is possible to make loans that must be repaid within the course of a single transaction

Frontrunning

Re-entrancy

DAO disaster

- Decentralized Autonomous Organization
  - "Like a VC fund" but decentralized
  - Implementation: a contract that controls money, and directs its disbursement according to "shareholder votes"
  - Shareholders buy in, pool their ETH (sending to contract)
  - Then vote on investments, which are made together
  - Users can "split" a DAO

“The DAO”

```
function splitDAO(
  uint _proposalID,
  address _newCurator
) noEther onlyTokenholders returns (bool _success) {

  ...
  // XXXXX Move ether and assign new Tokens.  Notice how this is done first!
  uint fundsToBeMoved =
      (balances[msg.sender] * p.splitData[0].splitBalance) /
      p.splitData[0].totalSupply;
  if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender)
== false) // XXXXX This is the line the attacker wants to run more than once
      throw;

  ...
  // Burn DAO Tokens
  Transfer(msg.sender, 0, balances[msg.sender]);
  withdrawRewardFor(msg.sender); // be nice, and get his rewards
  // XXXXX Notice the preceding line is critically before the next few
  totalSupply -= balances[msg.sender]; // XXXXX AND THIS IS DONE LAST
  balances[msg.sender] = 0; // XXXXX AND THIS IS DONE LAST TOO
  paidOut[msg.sender] = 0;
  return true;
}
```

How to upgrade a contract?

How to upgrade a contract?

- Contracts are default immutable
- If there is no useful ongoing state, don't: just replace it
- If there is ongoing state (e.g., account balances) then:
  - Don't allow upgrades — and pray you got the code right
  - Call upgradeable/replaceable library code
  - Create a complex mechanism to transfer state from an old contract instance to a new contract instance

Future of Ethereum

- Proof of stake
- Rollups
- Sharding