Stablecoins and Oracles

Dan Robinson, Paradigm
Georgios Konstantopoulos, Paradigm
https://cs251.stanford.edu/

Stablecoins

Stablecoin:
a cryptocurrency designed to trade at a fixed price

Why stablecoins?

- Get the convenience, programmability, and/or censorship-resistance of a cryptocurrency like Bitcoin, without the price volatility
- Integrate real-world currencies into on-chain decentralized applications
  - Prediction markets
  - Decentralized exchanges
  - Borrowing and lending

USD stablecoins

- We’ll use USD stablecoins for our examples
  - Target price of 1 token = $1
- The same principles could be applied to create tokens that trade at any price:
  - Other currencies (EUR, RMB...)
  - Other assets (gold, stocks…)
  - Imaginary assets (temperature?)

Types of stablecoin

	Centralized	**Decentralized**
Collateralized	Custodial stablecoins	Synthetics
Uncollateralized	Central bank digital currency	Seigniorage shares

Custodial stablecoins

	Centralized	Decentralized
Collateralized	Custodial stablecoins	Synthetics
Uncollateralized	Central bank digital currency	Seigniorage shares

Custodial stablecoins

Examples	Backing	Peg Mechanism	Risks
USDC, USDT	Dollars in a bank account somewhere	Issuance and redemption	Counterparty risk, regulatory risk

Custodial stablecoins

```
/**
 * @dev Function to mint tokens
 * @param _to The address that will receive the minted tokens.
 * @param _amount The amount of tokens to mint. Must be less than or equal
 * to the minterAllowance of the caller.
 * @return A boolean that indicates if the operation was successful.
 */
function mint(address _to, uint256 _amount)
    external
    whenNotPaused
    onlyMinters
    notBlacklisted(msg.sender)
    notBlacklisted(_to)
    returns (bool)
{
    require(_to != address(0), "FiatToken: mint to the zero address");
    require(_amount > 0, "FiatToken: mint amount not greater than 0");

    uint256 mintingAllowedAmount = minterAllowed[msg.sender];
    require(
        _amount <= mintingAllowedAmount,
        "FiatToken: mint amount exceeds minterAllowance"
    );

    totalSupply_ = totalSupply_.add(_amount);
    balances[_to] = balances[_to].add(_amount);
    minterAllowed[msg.sender] = mintingAllowedAmount.sub(_amount);
    emit Mint(msg.sender, _to, _amount);
    emit Transfer(address(0), _to, _amount);
    return true;
}
```

Custodial stablecoins

```
    /**
     * @dev Function to mint tokens
     * @param _to The address that will receive the minted tokens.
     * @param _amount The amount of tokens to mint. Must be less than or equal
     * to the minterAllowance of the caller.
     * @return A boolean that indicates if the operation was successful.
     */
    function mint(address _to, uint256 _amount)
        external
        whenNotPaused
        onlyMinters
        notBlacklisted(msg.sender)
        notBlacklisted(_to)
        returns (bool)
    {
        require(_to != address(0), "FiatToken: mint to the zero address");
        require(_amount > 0, "FiatToken: mint amount not greater than 0");

        uint256 mintingAllowedAmount = minterAllowed[msg.sender];
        require(
            _amount <= mintingAllowedAmount,
            "FiatToken: mint amount exceeds minterAllowance"
        );

        totalSupply_ = totalSupply_.add(_amount);
        balances[_to] = balances[_to].add(_amount);
        minterAllowed[msg.sender] = mintingAllowedAmount.sub(_amount);
        emit Mint(msg.sender, _to, _amount);
        emit Transfer(address(0), _to, _amount);
        return true;
    }
```

Custodial stablecoins

```
/**
 * @dev Function to mint tokens
 * @param _to The address that will receive the minted tokens.
 * @param _amount The amount of tokens to mint. Must be less than or equal
 * to the minterAllowance of the caller.
 * @return A boolean that indicates if the operation was successful.
 */
function mint(address _to, uint256 _amount)
    external
    whenNotPaused
    onlyMinters
    notBlacklisted(msg.sender)
    notBlacklisted(_to)
    returns (bool)
{
    require(_to != address(0), "FiatToken: mint to the zero address");
    require(_amount > 0, "FiatToken: mint amount not greater than 0");

    uint256 mintingAllowedAmount = minterAllowed[msg.sender];
    require(
        _amount <= mintingAllowedAmount,
        "FiatToken: mint amount exceeds minterAllowance"
    );

    totalSupply_ = totalSupply_.add(_amount);
    balances[_to] = balances[_to].add(_amount);
    minterAllowed[msg.sender] = mintingAllowedAmount.sub(_amount);
    emit Mint(msg.sender, _to, _amount);
    emit Transfer(address(0), _to, _amount);
    return true;
}
```

Central bank digital currency

	Centralized	Decentralized
Collateralized	Custodial stablecoins	Synthetics
Uncollateralized	Central bank digital currency	Seigniorage shares

Central bank digital currency

Examples	Backing	Peg Mechanism	Risks
None	By fiat	Issuance and redemption	Government control

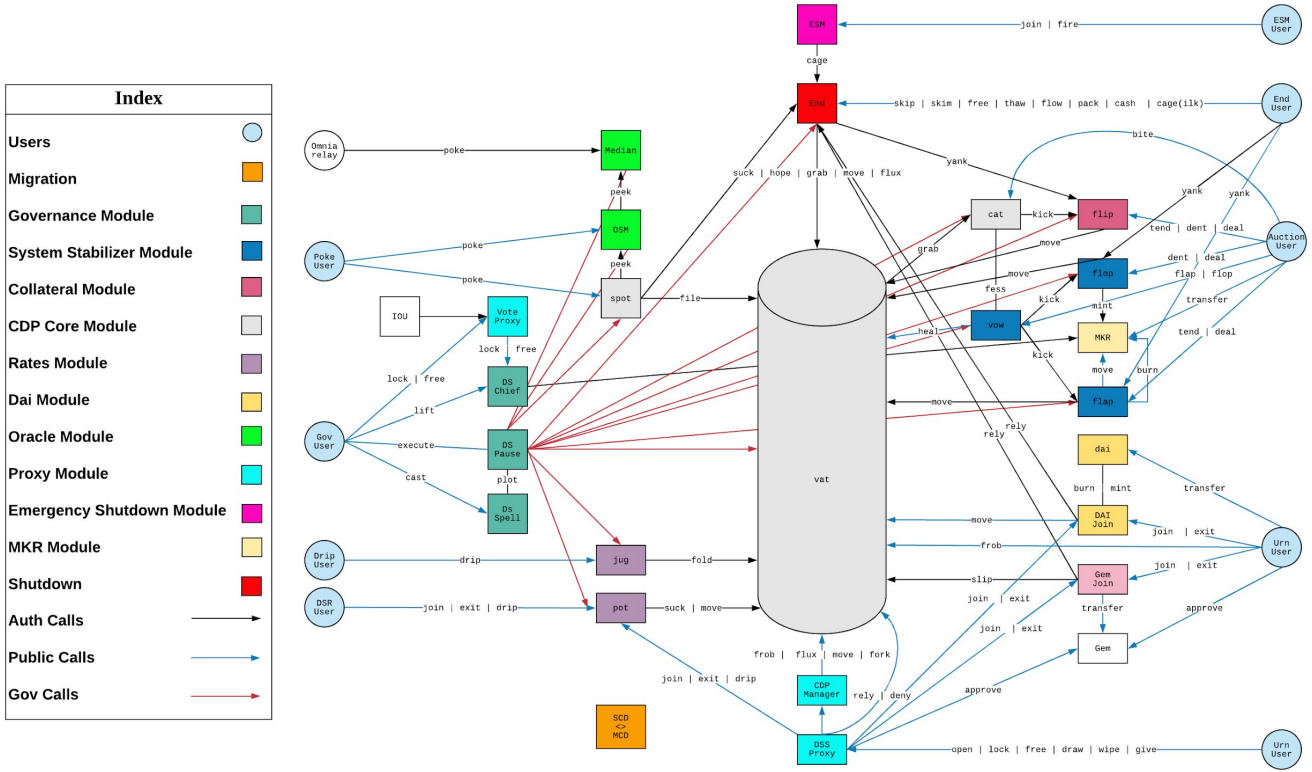
Synthetics

	Centralized	Decentralized
Collateralized	Custodial stablecoins	Synthetics
Uncollateralized	Central bank digital currency	Seigniorage shares

Synthetics

Examples	Backing	Peg Mechanism	Risks
Maker (DAI), BitShares	Native cryptocurrencies (ETH, BTC...)	Interest rate	Liquidation cascade, oracle dependency

Synthetics – Maker



Synthetics – Maker

```
// --- CDP Manipulation ---
function frob(bytes32 i, address u, address v, address w, int dink, int dart) external note {
    // system is live
    require(live == 1, "Vat/not-live");

    Urn memory urn = urns[i][u];
    Ilk memory ilk = ilks[i];
    // ilk has been initialised
    require(ilk.rate != 0, "Vat/ilk-not-init");

    urn.ink = add(urn.ink, dink);
    urn.art = add(urn.art, dart);
    ilk.Art = add(ilk.Art, dart);

    int dtab = mul(ilk.rate, dart);
    uint tab = mul(ilk.rate, urn.art);
    debt     = add(debt, dtab);
```

Synthetics – Maker

```
// --- CDP Manipulation ---
function frob(bytes32 i, address u, address v, address w, int dink, int dart) external note {
    // system is live
    require(live == 1, "Vat/not-live");

    Urn memory urn = urns[i][u];
    Ilk memory ilk = ilks[i];
    // ilk has been initialised
    require(ilk.rate != 0, "Vat/ilk-not-init");

    urn.ink = add(urn.ink, dink);
    urn.art = add(urn.art, dart);
    ilk.Art = add(ilk.Art, dart);

    int dtab = mul(ilk.rate, dart);
    uint tab = mul(ilk.rate, urn.art);
    debt     = add(debt, dtab);
```

?!?

Synthetics – Minting

Alice's Wallet		
Token	**Balance**	**USD value**
ETH	1	$300
DAI	0	$0

Alice's Vault		
Token	**Balance**	**USD value**
ETH	0	$0
DAI	0	$0

Alice wants to use Maker to get leverage on ETH

Synthetics – Minting

Alice's Wallet		
Token	**Balance**	**USD value**
ETH	0	$0
DAI	0	$0

Alice's Vault		
Token	**Balance**	**USD value**
ETH	1	$300
DAI	0	$0

Alice deposits 1 ETH into her Maker vault

Synthetics – Minting

Alice's Wallet		
Token	**Balance**	**USD value**
ETH	0	$0
DAI	200	$200

Alice's Vault		
Token	**Balance**	**USD value**
ETH	1	$300
DAI	-200	-$200

Alice uses her vault to mint 200 Dai to her wallet

Synthetics – Minting

Alice's Wallet		
Token	**Balance**	**USD value**
ETH	0.66	$200
DAI	0	$0

Alice's Vault		
Token	**Balance**	**USD value**
ETH	1	$300
DAI	-200	-$200

Alice trades her 200 DAI to Bob for 0.66 ETH

Synthetics – Minting

Alice's Wallet		
Token	**Balance**	**USD value**
ETH	0.66	$200
DAI	0	$0

Alice's Vault		
Token	**Balance**	**USD value**
ETH	1	$300
DAI	-200	-$200

Now Alice has levered up her exposure to ETH, and 200 new DAI is out there in the world

Synthetics – Stabilization

Alice's Vault		
Token	**Balance**	**USD value**
ETH	1	$300
DAI	-200	-$200

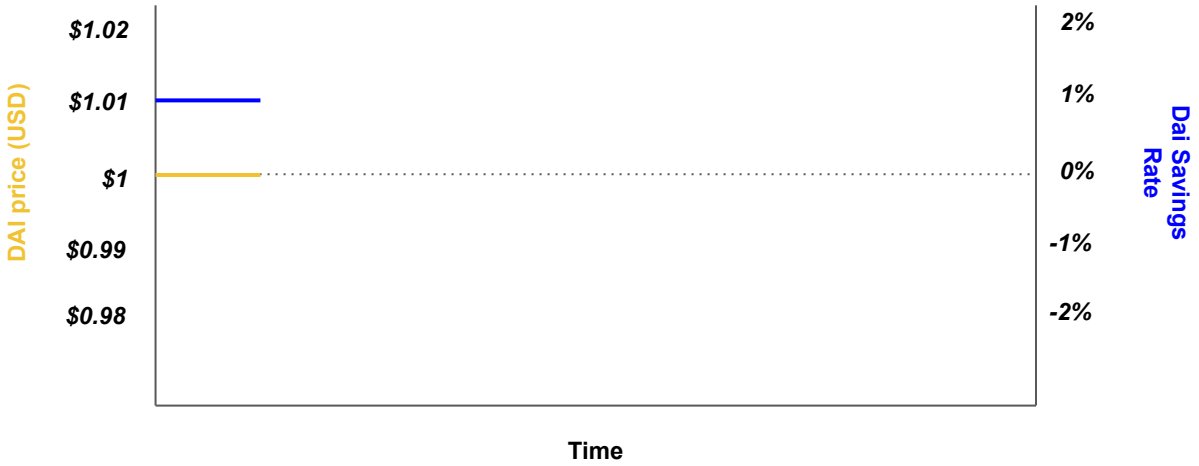
*Alice pays a **stability fee** as interest for borrowing DAI. Most of this stability fee goes to DAI holders through a mechanism called the **DAI Savings Rate (DSR)**. Part of it goes to the MKR token that governs the protocol.*

Synthetics – Stabilization

Alice's Vault, at time T+1		
Token	**Balance**	**USD value**
ETH	1	$300
DAI	-201	-$201

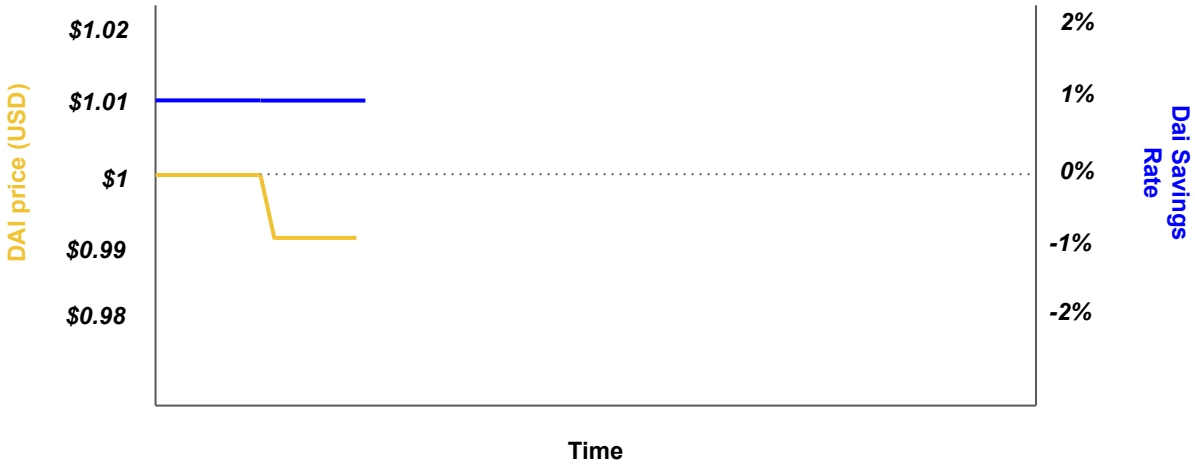
*Alice pays a **stability fee** as interest for borrowing DAI. Most of this stability fee goes to DAI holders through a mechanism called the **DAI Savings Rate (DSR)**. Part of it goes to the MKR token that governs the protocol.*

Synthetics – Stabilization



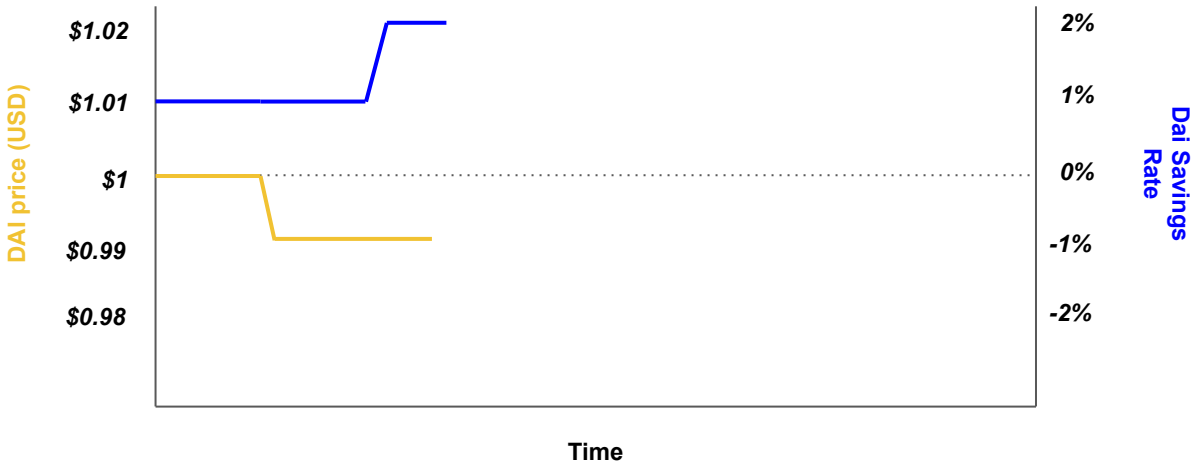
The stability fee and DSR are raised when DAI is trading below $1 (to discourage borrowing and encourage DAI holding), and lowered when DAI is trading above $1

Synthetics – Stabilization



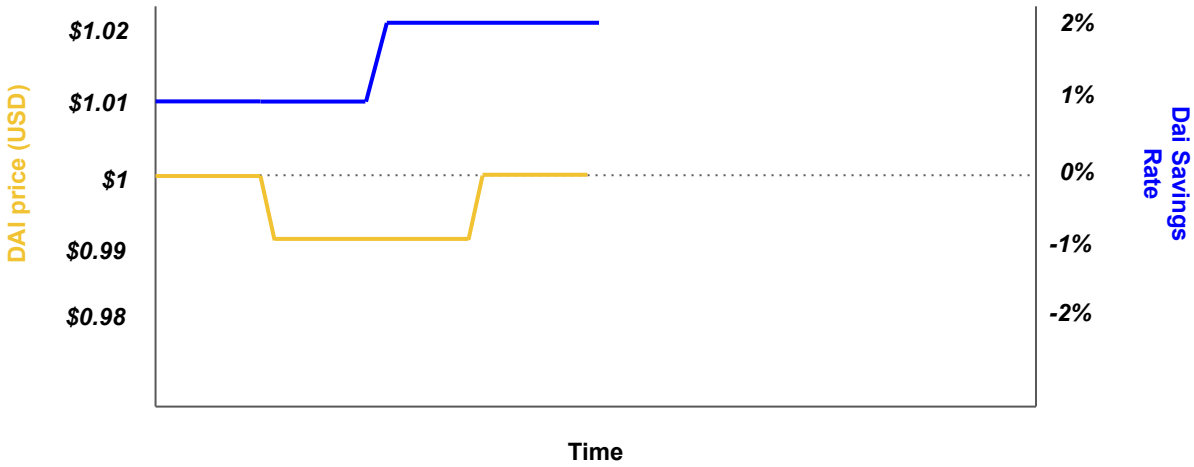
When the DAI price falls below $1...

Synthetics – Stabilization



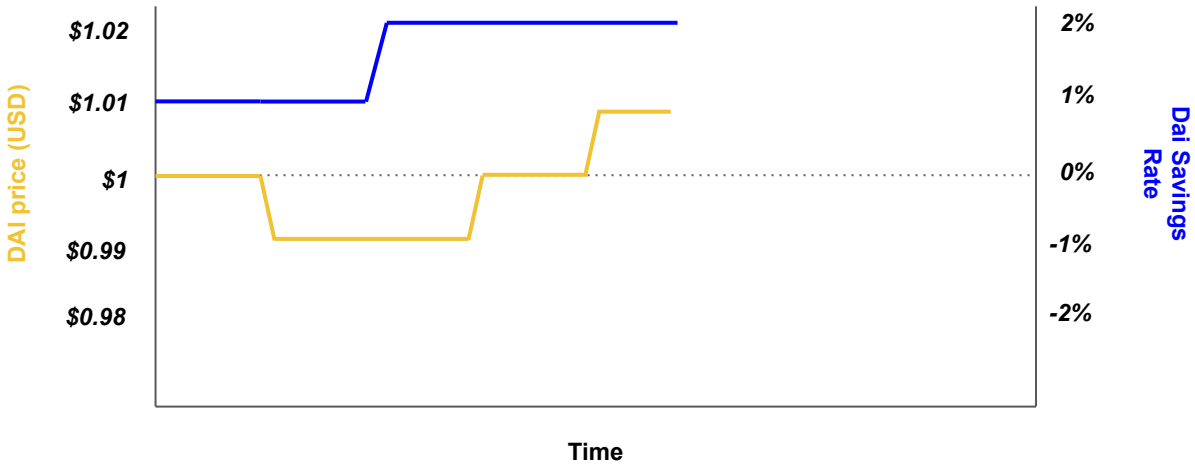
...the DSR (and stability fee) are raised to encourage DAI holding...

Synthetics – Stabilization



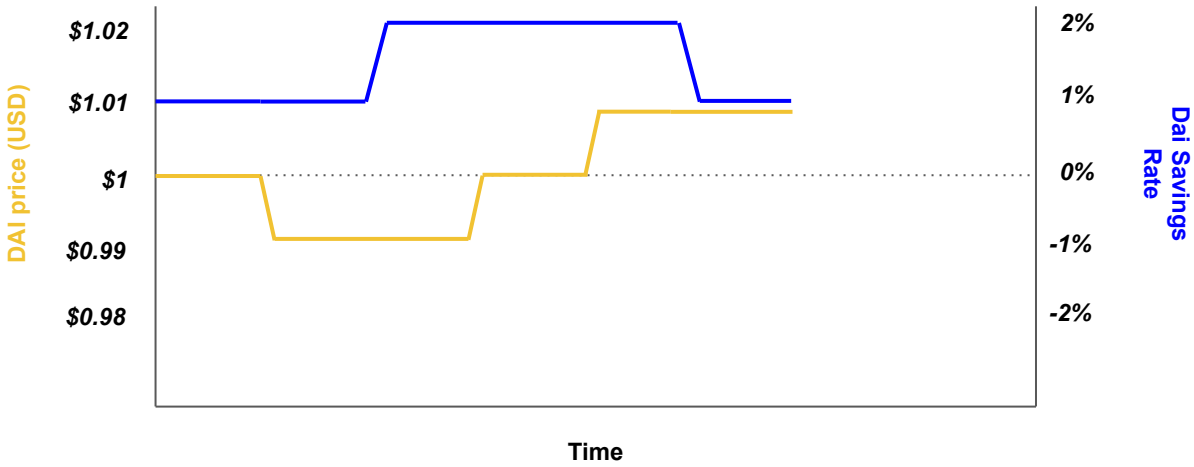
...causing the peg to be restored.

Synthetics – Stabilization



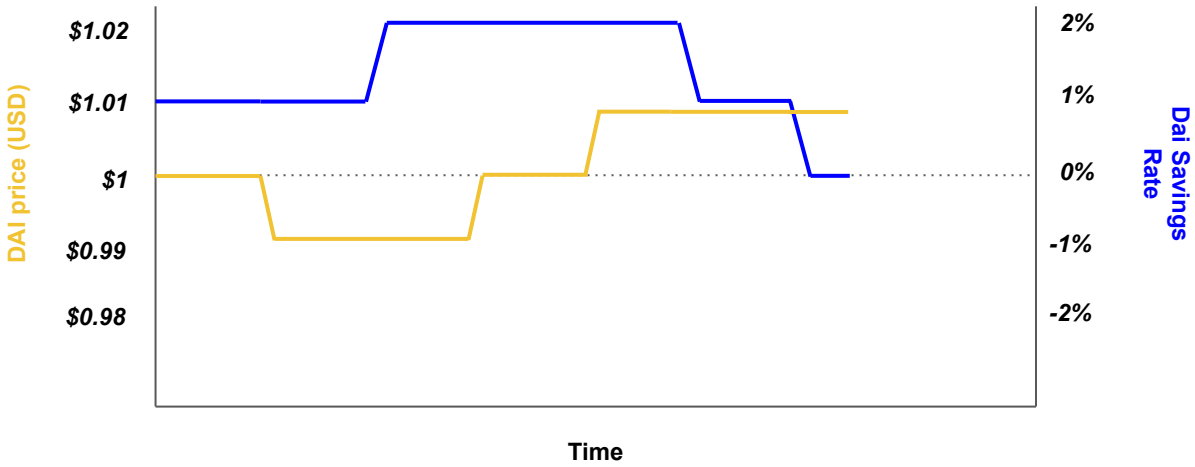
If DAI trades above $1...

Synthetics – Stabilization



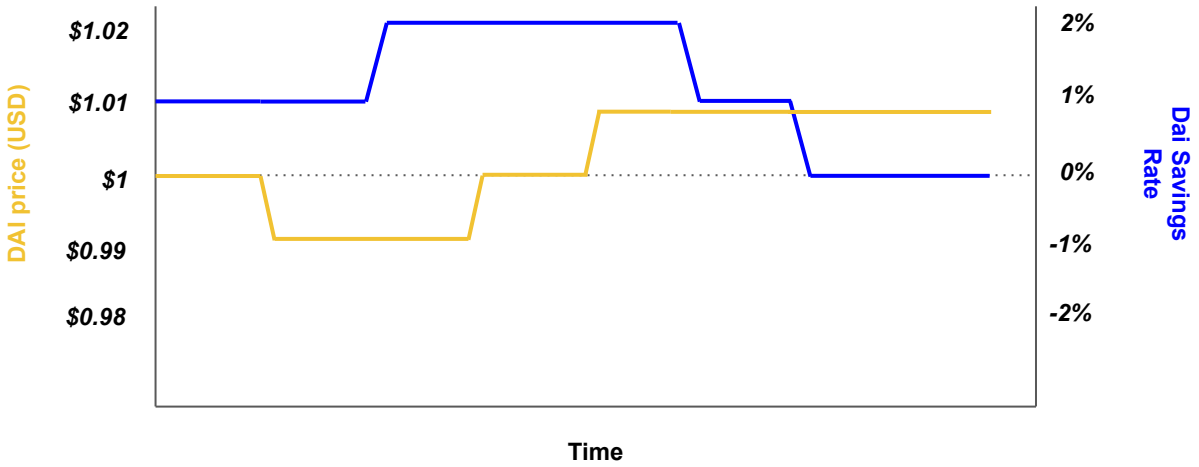
...the DSR is lowered...

Synthetics – Stabilization



...and continues to be lowered until the peg is restored...

Synthetics – Stabilization



...hopefully before the DSR and stability fee hit the zero lower bound.

Synthetics – Liquidation

Alice's Vault		
Token	**Balance**	**USD value**
ETH	1	$300
DAI	-200	-$200

Alice's vault is 150% collateralized, since it has $300 of collateral and $200 of debt

Synthetics – Liquidation

Alice's Vault		
Token	**Balance**	**USD value**
ETH	1	$298
DAI	-200	-$200

*If the price of ETH falls to $298, Alice is only 149% collateralized, which means her vault can be **liquidated***

Synthetics – Liquidation

Alice's Vault		
Token	**Balance**	**USD value**
ETH	0	$0
DAI	98	$98

In liquidation, the protocol auctions off Alice's ETH to repay her DAI debt.
She gets any extra DAI from the sale, minus fees (to MKR holders)

Seigniorage shares

	Centralized	Decentralized
Collateralized	Custodial stablecoins	Synthetics
Uncollateralized	Central bank digital currency	Seigniorage shares

Seigniorage shares

Examples	Backing	Peg Mechanism	Risks
● Basis ● Maker (backstop)	Confidence	Supply expansion and contraction	Death spiral, oracle dependency

Seigniorage shares – MKR backstop

Alice's Vault		
Token	**Balance**	**USD value**
ETH	1	$298
DAI	-200	-$200

Recall that when Alice's Maker vault had less than 150% collateral, the ETH was auctioned off

Seigniorage shares – MKR backstop

Alice's Vault		
Token	**Balance**	**USD value**
ETH	0	$0
DAI	-50	-$50

*Suppose ETH's price drops so sharply in price that Alice's ETH is only sold for 150 DAI, which is not enough to repay her 200 DAI debt. The protocol now has a **deficit**—there is 50 unbacked DAI out there*

Seigniorage shares – MKR backstop

Alice's Vault		
Token	**Balance**	**USD value**
ETH	0	$0
DAI	0	$0

The protocol mints new MKR tokens and auctions them off for 50 DAI, remedying the deficit. Recall that MKR tokens earn fees during normal operation of the protocol

Takeaways

- Many of these concepts have parallels in traditional monetary economics
  - Zero lower bound
  - Speculative attacks
  - Crisis of confidence
- Even decentralized stablecoins depend on price oracles
  - Georgios is covering that next!

Oracles

Background

- A blockchain cannot access data outside of its state (e.g. ETHUSD price, the weather today etc.)
- Complex use cases require non-native data:
  - Finance: prices, insurance
  - Random number generation
  - Blockchain interoperability: bitcoin headers on ethereum
  - IoT: temperature, geolocation data etc.

How do you import non-native data to a blockchain? Oracles!

The oracle problem

- “Good data” vs “bad data”: subjective
- How do you penalize offenders? Circular argument
    - “who guards the guards”

The oracle problem

- “Good data” vs “bad data”: subjective
- How do you penalize offenders? Circular argument
  - “who guards the guards”

< end of talk >

Oracles in Decentralized Finance (DeFi)

- Lending
- Synthetics and stablecoins
- Leverage

Oracles in Decentralized Finance (DeFi)

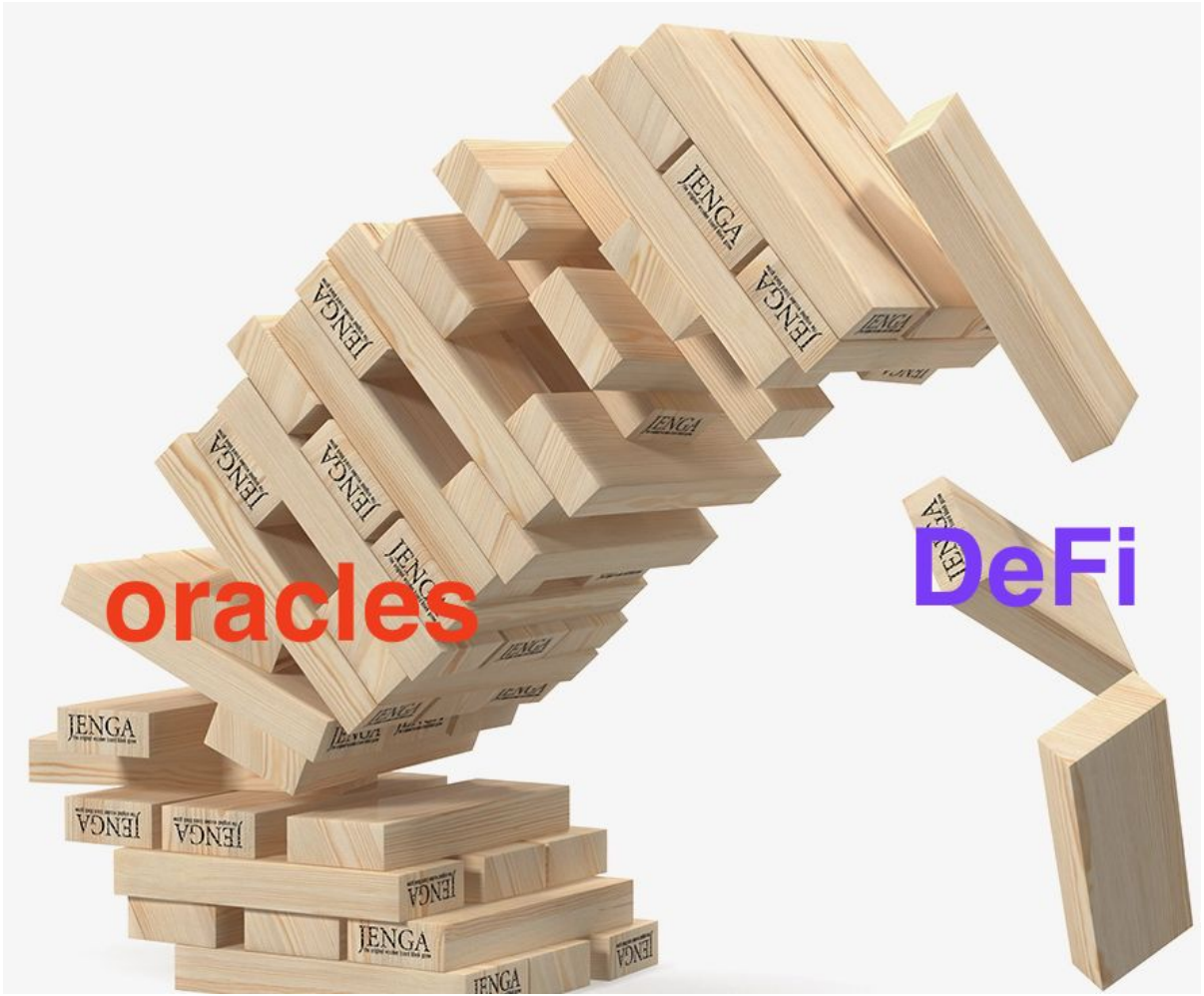
- Lending
- Synthetics and stablecoins
- Leverage

Oracles determine your maximum allowed debt:

*max debt = collateral * price * threshold*

*E.g. Max DAI debt in Maker for 1 ETH @ $150: 1 ETH * $150 * ⅔ =* ***100 DAI***

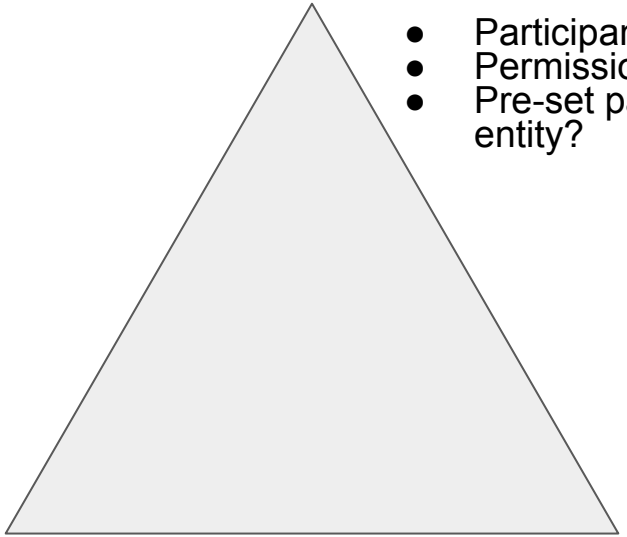
if the price changes (e.g. 1 ETH = $140) and your max debt (93.33 DAI) is less than your current debt (100 DAI) **you get liquidated and pay a penalty**


oracles
DeFi
JENGA

The Oracle Trilemma

Decentralized

- Participant set size?
- Permissioned / permissionless entry?
- Pre-set participants by privileged entity?



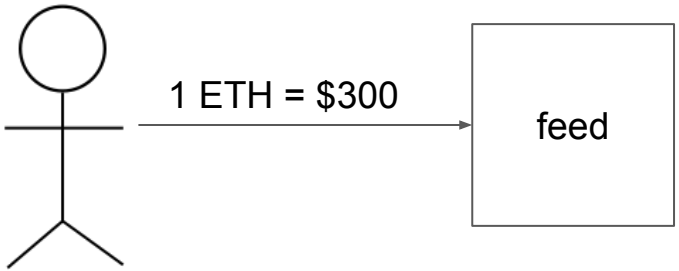
High frequency/accuracy

- How frequently are new values published?
- Does the value at publication time match the off-chain value?
- Can someone challenge a bad entry in time?

High corruption cost:

- Cost of corruption < Profit from corruption?

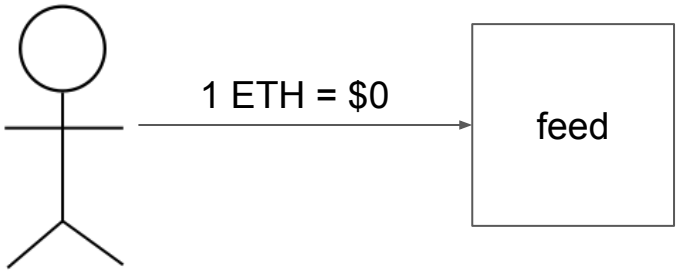
Strawman: Single oracle



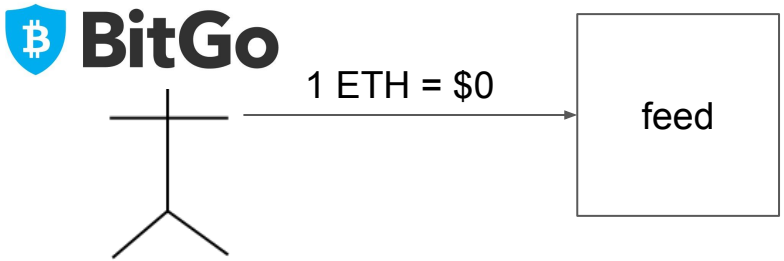
Strawman: Single oracle

```
contract Oracle {
    address oracle;
    uint256 public ethusd;

    constructor() {
        oracle = msg.sender;
    }

    function update(uint256 _ethusd) external {
        require(msg.sender == oracle, "auth error");
        ethusd = _ethusd;
    }
}
```

...is trivial to corrupt

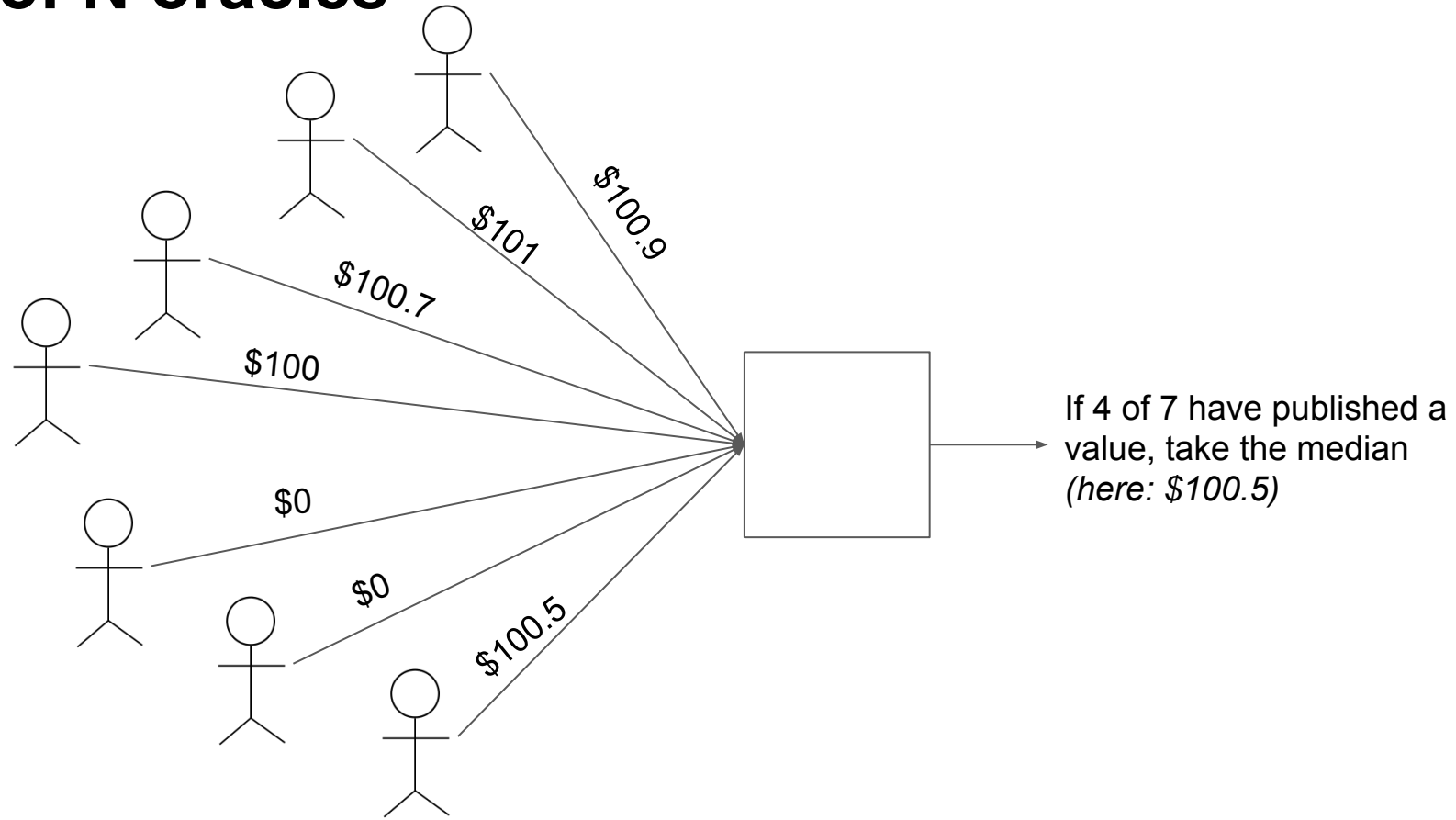


...or not? Skin in the game matters

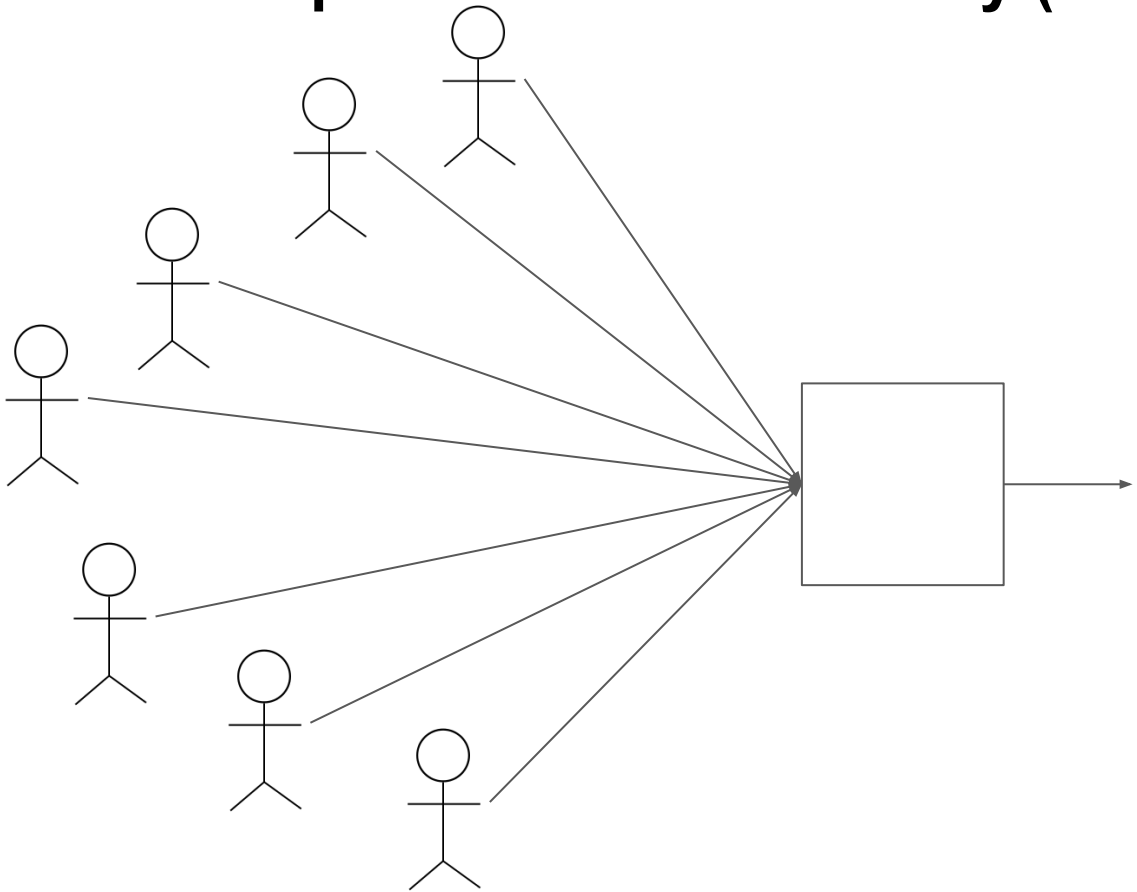


worse is better, keep it simple etc.

M of N oracles



...need to corrupt M+1 to violate safety (or N-M+1 for liveness)



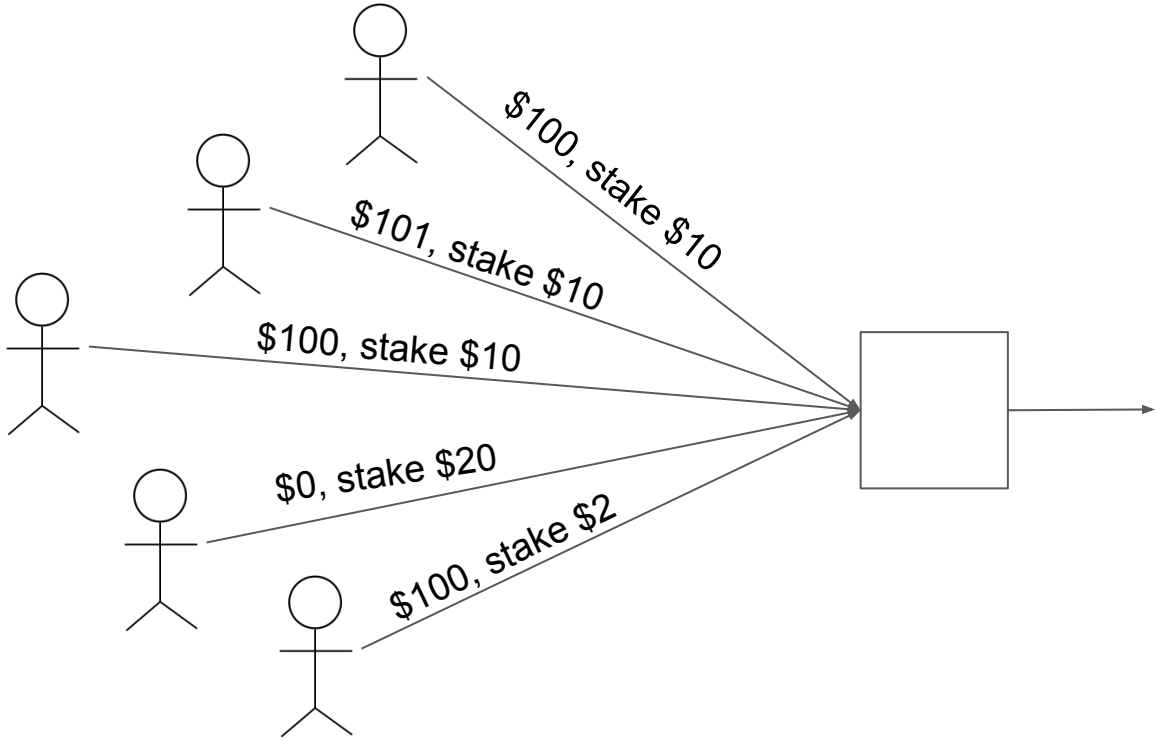
If 4 of 7 are corrupt → median can be manipulated

If 3 of 7 are corrupt → DoS

Who chooses the oracles?

- Static: Set at deploy time, can never change
- Dynamic: Privileged entity that can add / remove oracles at will
- **Both cases: permissioned**
- **Can there be an oracle system with an open participant set?**

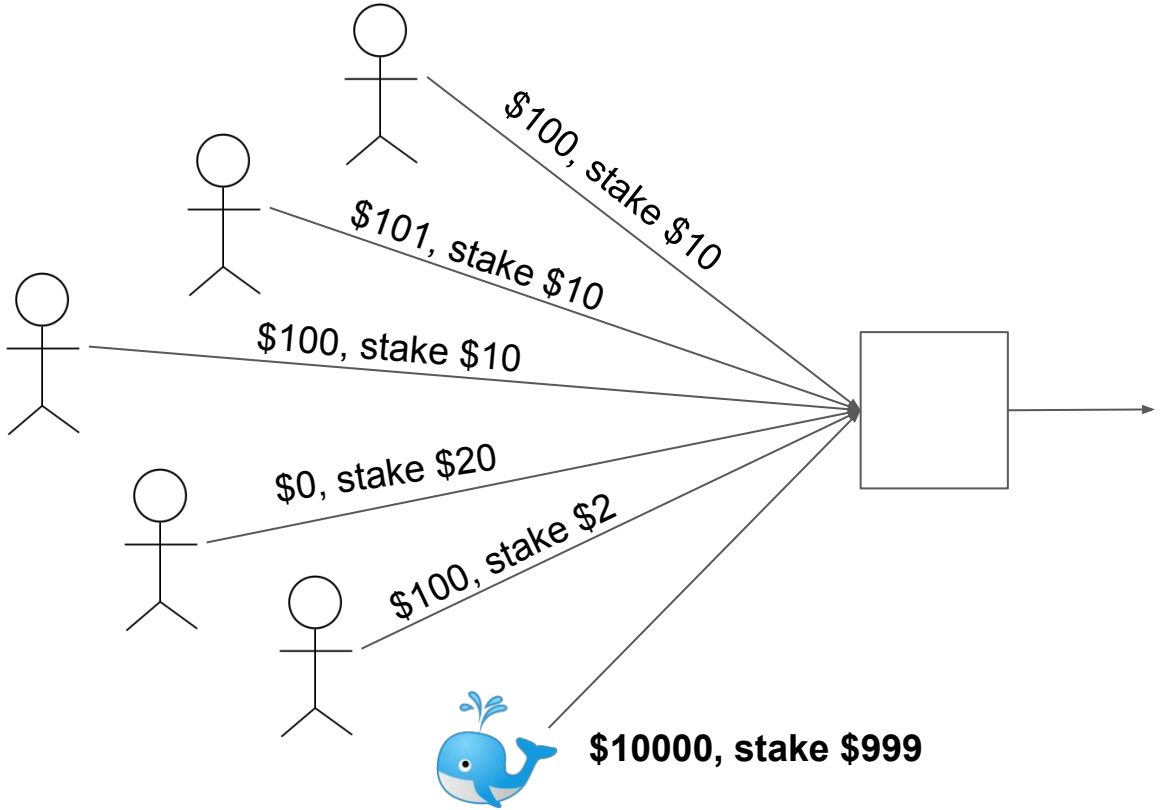
Schelling oracles (1 vote = $1)



After 2 hours & if at least $50 has voted, take the value with most
$ staked (+ slash everyone that didn't vote close to it)

...are subject to whale manipulation + slow to allow enough stake to vote*

**can appeal to re-run the vote if result is not desired one*

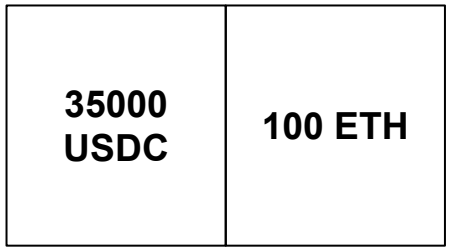


After 2 hours & if at least $50 has voted, take the value with most $ staked (+ slash everyone that didn't vote close to it)

Can we have oracles that do not require identity (or a proxy of it?) → Markets*!

**for assets that have an on-chain representation*

Strawman: Query on-chain price

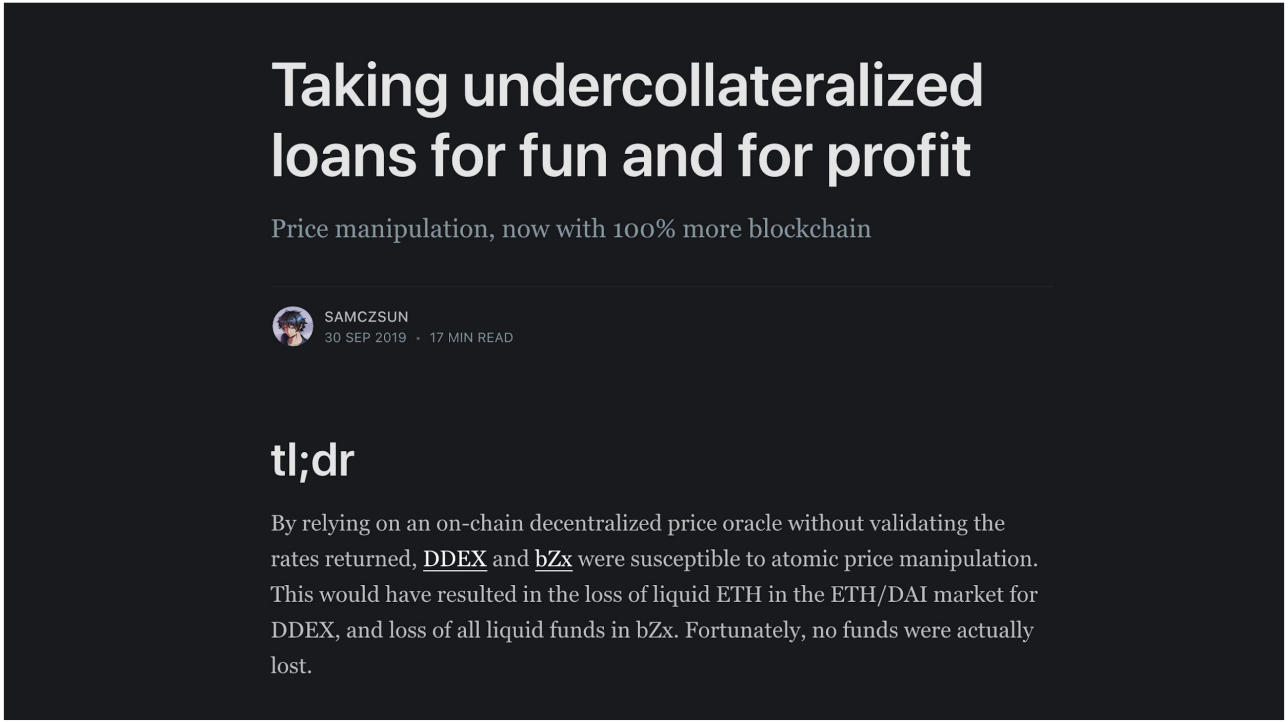


Reserves
(50%-50%)

ETHUSD = 35000 / 100 = $350

price1in0 = (supply0 / factor0) / (supply1 / factor1)

… is vulnerable to “sandwich” attacks



Taking undercollateralized loans for fun and for profit

Price manipulation, now with 100% more blockchain

SAMCZSUN
30 SEP 2019 • 17 MIN READ

tl;dr

By relying on an on-chain decentralized price oracle without validating the rates returned, DDEX and bZx were susceptible to atomic price manipulation. This would have resulted in the loss of liquid ETH in the ETH/DAI market for DDEX, and loss of all liquid funds in bZx. Fortunately, no funds were actually lost.

https://samczsun.com/taking-undercollateralized-loans-for-fun-and-for-profit

… is vulnerable to “sandwich” attacks

Alice’s Wallet		
Token	**Balance**	**USD value**
ETH	1	$350
USDC	35000	$35000

Alice’s Lending Vault		
Token	**Balance**	**USD value**
ETH	0	$0
USDC	0	0

100 ETH	**35000 USDC**

Alice wants to borrow USDC against her ETH.
The lending protocol uses the Uniswap ETH-USDC pair as a price oracle.*

**0 fees for simplicity*

… is vulnerable to “sandwich” attacks

Alice’s Wallet		
Token	**Balance**	**USD value**
ETH	0	$0
USDC	35000	$35000

Alice’s Lending Vault		
Token	**Balance**	**USD value**
ETH	1	$350
USDC	0	0

100 ETH	**35000 USDC**

*Alice deposits ETH into her vault. Normally, this would allow her to borrow up to $350 * ⅔ = $233 USDC.*

… is vulnerable to “sandwich” attacks

Alice’s Wallet		
Token	**Balance**	**USD value**
ETH	50	$17500
USDC	0	$0

Alice’s Lending Vault		
Token	**Balance**	**USD value**
ETH	1	$1400
USDC	0	0

50 ETH	**70000 USDC**

Alice buys 50 ETH on Uniswap inflating the Uniswap (!) price to $1400.
In other venues, the price is still $350.

… is vulnerable to “sandwich” attacks

Alice’s Wallet		
Token	**Balance**	**USD value**
ETH	50	$17500
USDC	933	$933

Alice’s Lending Vault		
Token	**Balance**	**USD value**
ETH	1	$1400
USDC	-933	-$933

*Atomically, Alice borrows ⅔ * 1400 = 933 USDC.*

50 ETH	**70000 USDC**

… is vulnerable to “sandwich” attacks

Alice’s Wallet		
Token	**Balance**	**USD value**
ETH	0	$0
USDC	35933	$35933

Alice’s Lending Vault		
Token	**Balance**	**USD value**
ETH	1	$350
USDC	-933	-$933

100 ETH	**35000 USDC**

Atomically, Alice restores Uniswap’s price

The lending protocol now has -$583 in debt and Alice made a $583 profit at no cost. Bad oracle.

Time Weighted Average Price

Problem:

Easy to manipulate prices in a short timescale

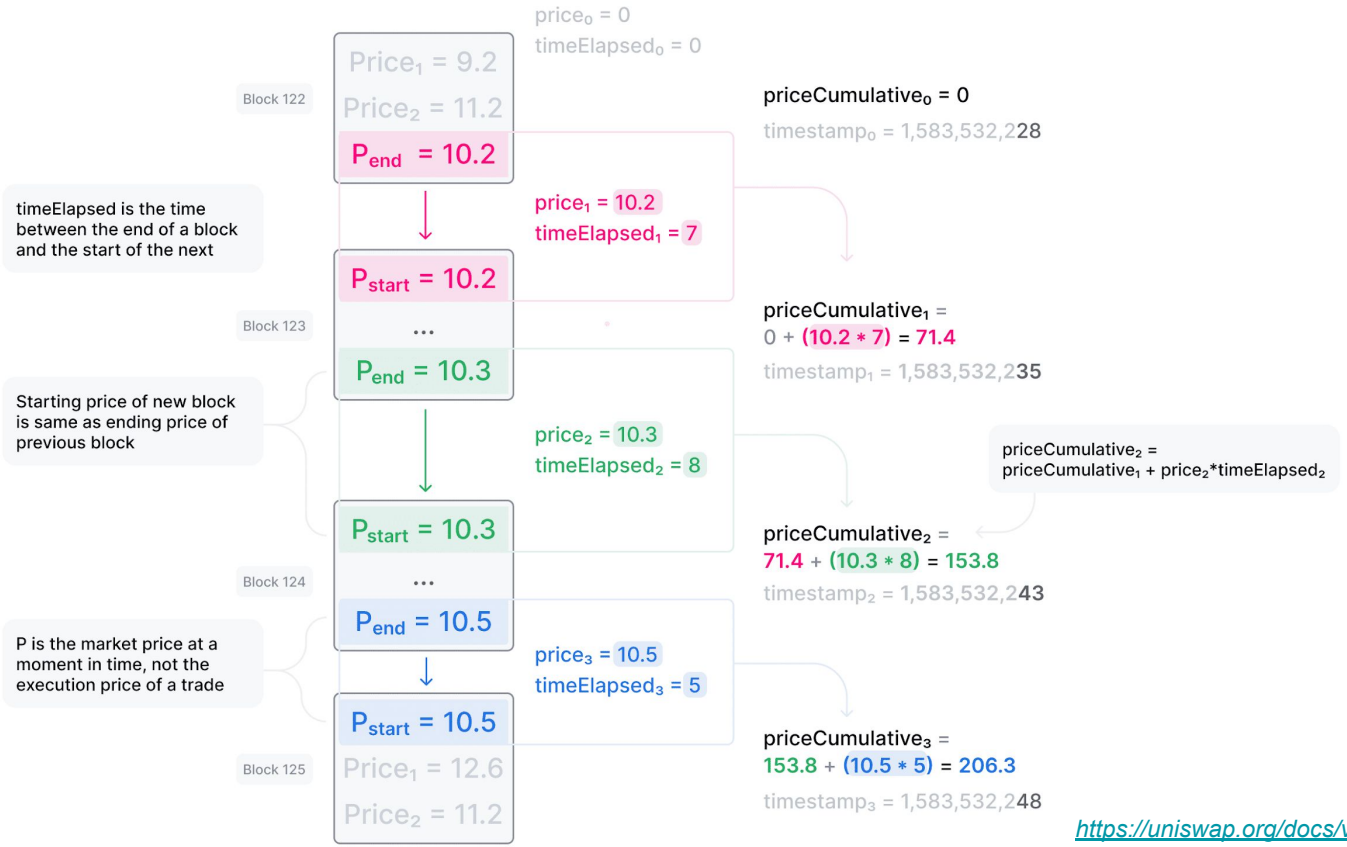
Solution

Store prices over time (e.g. 1 week) and average over them

Tradeoff

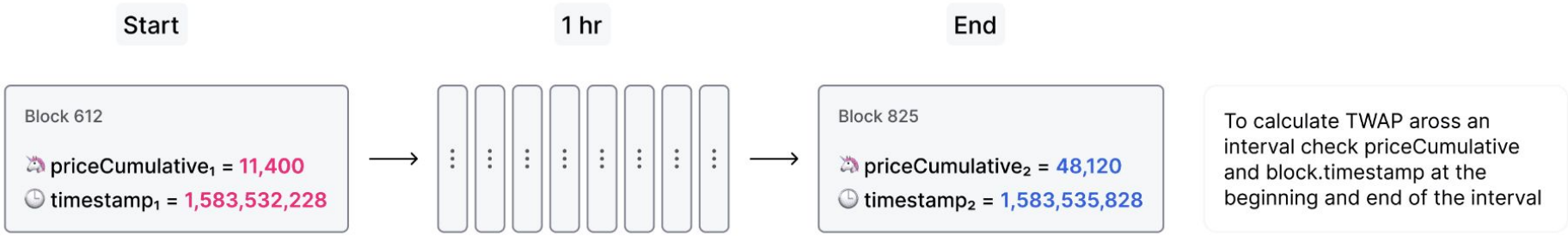
Security - frequency

Time Weighted Average Price



https://uniswap.org/docs/v2/core-concepts/oracles/

Time Weighted Average Price



$$\textbf{TWAP} = \frac{priceCumulative_2 - priceCumulative_1}{timestamp_2 - timestamp_1} = \frac{48{,}120 - 11{,}400}{1{,}583{,}535{,}828 - 1{,}583{,}532{,}228} = \textbf{10.2}$$

https://uniswap.org/docs/v2/core-concepts/oracles/

Order Books and Auctions

- **Idea: “In an efficient market, an unclaimed two-way price quote is an oracle”**[1, 2]
- Place buy/sell orders
- Mispriced orders get arbitraged away
- Price = remaining order price after T

Order Books and Auctions

Example (1 ETH = $350)

1 ETH + 350 USDC
(price = $350)

Initially, there's an "accurate" order on the orderbook.

https://nestprotocol.org/assets/pdf/ennestwhitepaper.pdf
https://twitter.com/danrobinson/status/1162750513521164288
https://github.com/keep-network/tbtc/issues/254

Order Books and Auctions

Example (1 ETH = $350)

10 ETH + 3000 USDC
(price = $300)

An attacker wants to manipulate the price down.

They fill it by buying 350 USDC for 1 ETH (or vice versa), and MUST provide a 10x bigger ETH bid with their new price proposal

Order Books and Auctions

Example (1 ETH = $350)

arb for $50 profit per ETH

10 ETH + 3000 USDC
(price = $300)

An arbitrageur notices it, buys 10 ETH for $300 each and sells for $350

Order Books and Auctions

Example (1 ETH = $350)

100 ETH + 35000 USDC
(price = $350)

...and puts up a larger bid.

1 hour passes with no new bids, auction ends, price gets set to $350

arbitrageur receives back the order (+ extra compensation)

Recap: Mapping the oracle design space

		Decentralized	Freq / Accuracy	Corruption Cost
identity	**1 signer**	Low	High	Low
	M-of-N signers	Medium	High	Medium
	Schelling game	Depends on token distribution	High	Depends on token distribution
markets	**Query on-chain price**	High	High	Low
	Uniswap TWAP	High	Configurable	Scales inversely with frequency
	Orderbook / Auction	High	Configurable	Scales inversely with frequency

Thank you for your attention!

@gakonst
georgios@paradigm.xyz

Appendix

Illiquid capital is… expensive

- Staking tokens = opportunity cost
- Yields must exceed risk-adjusted alternative yield sources (e.g. lending)[1]
- Schelling oracle
  - Pay correct voters by slashing wrong voters
  - Reward with native token
- Orderbook
  - Pay makers with % of liquidation fees
  - Reward with native token
- TWAP:
  - Piggybacks on Uniswap’s pre-existing trading activity.
  - LPs are paid by trading fees

Optimization: Reduce oracle operating costs

Oracles: Submit update every T

“Optimistic” oracle: Submit update each time there’s a dispute

1. User tries to draw $500 against 1 ETH (1 ETH = $350 in real world)
2. Somebody watching: “hey this is more than you’re allowed to have”
3. Oracle posts the value → cancels loan

Pros: Less gas paid by oracles

Cons: Synchrony assumption, additional watching infrastructure

Frontrunning aka Priority Gas Auction (PGA)

Broadcast transaction A with a **higher** gasPrice than already pending transaction B so that A gets mined **before** B

Good case: Maker Vault

1. Liquidation price: $140
2. Pending oracle price update to $138 @ gasPrice = 50gwei
3. Broadcast “repay debt” with gasPrice = 51gwei
4. Debt is repaid, saved from liquidation

Bad case: Synthetix

1. Frontrun ETH oracle updates:
   a. Price  → buy sETH / sell iETH
   b. Price  → buy iETH / sell sETH
2. Risk-free profit

(addressed in SIP-6, allows oracle to frontrun the frontrunner and burn their balance[1])

https://arxiv.org/abs/1904.05234
frontrun.me

Backrunning

Broadcast transaction A with a **lower** gasPrice than already pending transaction B so that A gets mined **after** B

Example 1: Liquidations

1. Oracle update in the mempool (decreases the price)
2. Positions become eligible for liquidation
3. Backrun the oracle update
4. Liquidation succeeds since it's included right after the oracle update

Example 2: AMM Arbitrages (non-oracle related)

1. Uniswap price at parity
2. 'Buy' trade gets submitted
3. Backrun with 'Sell'
4. Get arbitrage profit

https://github.com/ethereum/go-ethereum/issues/21350
https://explore.duneanalytics.com/public/dashboards/FFFpCKoE41bvFpESiyjUIBJfEMt4GoMFwcidNcAh

“Generalized” oracles

- Oracles are not just for fetching “static” values e.g market prices
- Can use oracles for minimizing their on-chain footprint...aka scaling!

e.g. instead of doing the computation, you could do the computation off-chain and publish its result → this is still an oracle about the state of a system!

How do you guarantee that the result is correct?

- Fraud proof within 7 days (or any other security param)
- Validity proof (SNARK / STARK etc.)

Trusted Execution Environments

TEEs (e.g. Intel SGX):

- Remote attestation:
  - Prove that a computation was done within a certain hardware version
- Trusted Hardware:
  - Data is fetched and “signed” by the enclave