L05: Smart Contracts & DeFi Engineering

Extended Slides – The Composability Paradox

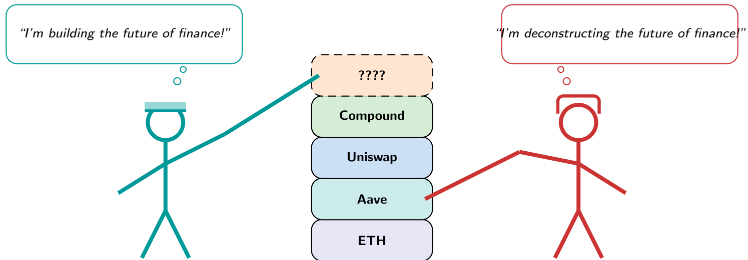
Digital Finance

What Will You Be Able to Do After This Lecture?

1. Explain EVM execution, opcode gas costs, and EIP-1559 fee market
2. Classify smart contract vulnerabilities and apply formal verification tools
3. Derive constant product AMM formula, compute impermanent loss
4. Trace flash loan attacks, calculate atomic arbitrage profit
5. Analyze MEV extraction and mitigation strategies
6. Model governance attacks, oracle cascades, composability risk

---

Six objectives: Solidity engineering (1–2), AMM mathematics (3), flash loan mechanics (4), MEV economics (5), and composability risk (6). This lecture combines formal derivations with working code and 11 data visualizations.



Permissionless composability: the same API that builds the tower lets anyone pull out the blocks.

How Does a Stack Machine Execute Your Smart Contract – and Why Does Every Instruction Have a Price?

Definition. The EVM is a deterministic stack machine. Given world state σ and transaction T:

$$\sigma' = \Upsilon(\sigma, T)$$

Gas model. Total gas consumed by a transaction:

$$G_{\text{total}} = G_{\text{intrinsic}} + \sum_i g(o_i) + G_{\text{memory}} + G_{\text{storage}}$$

Key opcode costs:

- `SSTORE` (storage write): 20,000 gas (cold) / 5,000 gas (warm)
- `SLOAD` (storage read): 2,100 gas (cold) / 100 gas (warm)
- `MSTORE` (memory write): 3 gas
- Storage is $\sim$7,000$\times$ more expensive than memory

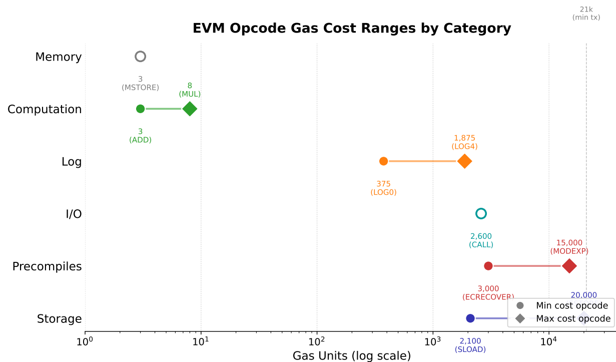
EIP-1559 base fee adjustment. After block n with gas used g_n and target g_{target}:

$$b_{n+1} = b_n \left(1 + \frac{1}{8} \cdot \frac{g_n - g_{\text{target}}}{g_{\text{target}}}\right)$$

Deflationary condition: Ethereum burns base fee. Net issuance is negative when burn $>$ staking rewards, i.e., when sustained demand keeps $g_n > g_{\text{target}}$.

Every EVM instruction has a gas price that reflects its computational and storage cost. Storage is 7,000x more expensive than memory – this single fact drives most Solidity optimization.

How Much Does Each Type of Computation Cost on Ethereum?



The gas barbell: costs cluster at two extremes.

- **Cheap end:** arithmetic (`ADD` 3 gas), stack (`PUSH` 3 gas), memory (`MSTORE` 3 gas)
- **Expensive end:** storage (`SSTORE` 20,000 gas), contract creation (`CREATE` 32,000 gas)
- The middle is nearly empty – costs are bimodal
- Optimization strategy: move computation from the right (storage) to the left (memory/stack)

Implication: gas-efficient contracts cache storage reads in memory variables and batch writes into a single operation.

Storage writes cost 7,000x more than memory operations. This is why Solidity optimization is fundamentally about minimizing state changes – every SSTORE is a small tax.

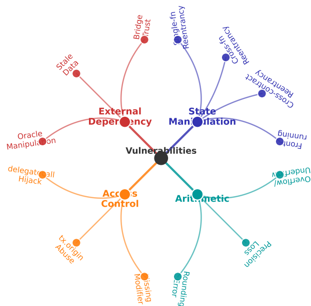
Can You Cut a Smart Contract's Gas Cost in Half with Three Tricks?

```python
1  # Gas optimization: storage vs memory patterns
2  def expensive_loop(balances, users):
3      """BAD: reads storage in every iteration."""
4      total = 0
5      for u in users:
6          total += balances[u]  # SLOAD each time: 2100 gas * N
7      return total
8
9  def cheap_loop(balances, users):
10     """GOOD: cache in memory, single storage write."""
11     cache = {u: balances[u] for u in users} # N SLOADs
12     total = sum(cache.values()) # memory ops: 3 gas * N
13     return total
14     # Savings: (2100 - 3) * (N-1) gas for N users
```

Three rules: read storage once, compute in memory, write storage once. At 100 users this saves 207,000 gas – the difference between a $5 and a $9 transaction.

What Are the Species in the Smart Contract Vulnerability Zoo?

Smart Contract Vulnerability Taxonomy



Four vulnerability families:

- **State manipulation:** reentrancy, front-running, state inconsistency
- **Arithmetic:** overflow/underflow, precision loss, rounding errors
- **Access control:** unprotected functions, tx.origin confusion, missing modifiers
- **External dependency:** oracle manipulation, flash loan attacks, composability exploits

The dendrogram reveals a key pattern: most novel exploits are *hybrids* combining two or more families (e.g., flash loan + oracle manipulation).

Over 80 percent of exploits fall into four categories: state manipulation, arithmetic, access control, and external dependency. The taxonomy is a map of the attack surface.

(c) Joerg Osterrieder 2026

Why Is Reentrancy a Cycle in the Call Graph – and How Do You Prove Its Absence?

Call graph. Model contract execution as a directed graph $G = (V, E)$:

- V = set of functions (internal and external)
- $E \subseteq V \times V$ = call edges between functions

Partition. Let $W \subseteq V$ be *state-write* functions and $X \subseteq E$ be *external call* edges.
Reentrancy condition. A reentrancy vulnerability exists iff:

$$\exists \text{ cycle } C \text{ in } G \text{ such that } \exists\, e \in C \cap X,\ \exists\, w \in C \cap W : \quad e \to_C w$$

That is, an external call edge e in the cycle reaches a state-write function w via the cycle path.
Safety property. A contract is reentrancy-safe iff:

$$\text{Safe}(G) \iff \forall \text{ cycles } C : \neg(\exists\, e \in C \cap X,\ \exists\, w \in C \cap W : \ e \to_C w)$$

Checks-Effects-Interactions (CEI) as a topological constraint: impose a partial order where all edges to W (state writes) precede all edges in X (external calls). This eliminates all vulnerable cycles.
Complexity: CEI violations are detectable in $O(|V| + |E|)$ via depth-first search on the call graph.

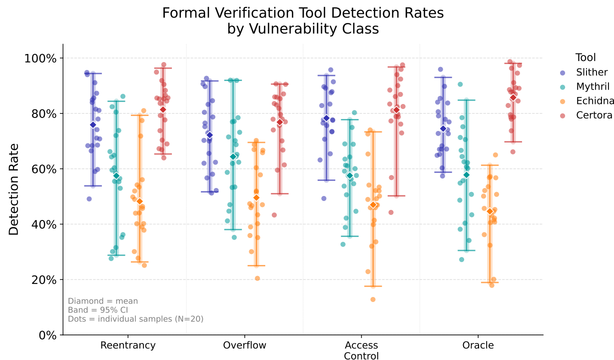
Reentrancy reduces to a graph problem: find cycles in the call graph that contain both an external call and a subsequent state write. CEI enforces a topological constraint: all writes before all external calls. Static analyzers exploit this for O(V+E) detection.

Can You Build a Reentrancy Detector in 15 Lines of Python?

```
1  # Reentrancy detector: find CALL-before-SSTORE violations
2  def detect_reentrancy(functions):
3      """Check if any function does external call before state write."""
4      vulns = []
5      for name, ops in functions.items():
6          last_write = -1
7          first_call = len(ops)
8          for i, op in enumerate(ops):
9              if op.startswith("SSTORE"):
10                 last_write = max(last_write, i)
11             if op.startswith("CALL") and i < first_call:
12                 first_call = i
13         if first_call < last_write: # CEI violation
14             vulns.append((name, first_call, last_write))
15     return vulns
16
17 fns = {"withdraw": ["SLOAD", "DUP", "PUSH", "CALL", "POP", "SSTORE"]}
18 print(detect_reentrancy(fns)) # [('withdraw', 3, 5)]
```

Real tools (Slither, Mythril) use symbolic execution, but the core insight is simple: if CALL comes before SSTORE in any execution path, reentrancy is possible.

How Good Are Automated Tools at Finding Smart Contract Bugs?



Verification tool landscape:

- **Slither:** static analysis, fast (<1 min), catches known patterns, high false-positive rate
- **Mythril:** symbolic execution, medium speed, deeper coverage of execution paths
- **Certora:** formal verification, slowest, proves properties mathematically but requires manual spec writing
- **Echidna:** fuzzing, randomized input testing, finds edge cases missed by static tools

Key gap: no tool tests cross-protocol composability – the largest and fastest-growing attack surface in DeFi.

No single tool catches all vulnerabilities. Slither finds known patterns fast; Certora proves properties formally. The gap: none test cross-protocol composability – the biggest attack surface in DeFi.

(c) Joerg Osterrieder 2026

Why Does x Times y Equals k Actually Work as a Market?

Constant product invariant. A Uniswap-style AMM holds reserves (x, y) of two tokens satisfying:

$$x \cdot y = k \qquad \text{(invariant, preserved after every trade)}$$

Marginal price. The instantaneous price of token A in terms of token B:

$$P_A = \frac{y}{x} = \frac{dy}{dx}\bigg|_{xy=k} \quad \Rightarrow \quad \text{price moves with every trade}$$

Trade execution. A trader swaps Δx of token A (with fee ϕ):

$$\Delta y = \frac{y \cdot \Delta x \cdot (1-\phi)}{x + \Delta x \cdot (1-\phi)}$$

Slippage. The price impact of a trade of size Δx relative to reserves x:

$$\text{Slippage} = 1 - \frac{\Delta y / \Delta x}{y/x} = 1 - \frac{x \cdot (1-\phi)}{x + \Delta x \cdot (1-\phi)} \approx \frac{\Delta x}{x} \quad \text{for small trades}$$

Slippage grows linearly with trade size relative to pool depth – deep pools have lower slippage.

x times y equals k is the simplest possible market maker. The convexity of the pricing function protects liquidity providers from large trades – but creates MEV opportunities for bots who split trades optimally.

How Much Do Liquidity Providers Actually Lose – and Is It Really "Impermanent"?

Setup. An LP deposits equal value of tokens A and B at price ratio $P_0 = y_0/x_0$. After the price moves to $P_1 = r \cdot P_0$ (where r is the price ratio change):

Rebalanced pool. The AMM adjusts reserves to maintain $x \cdot y = k$:

$$x_1 = \frac{x_0}{\sqrt{r}}, \qquad y_1 = y_0 \cdot \sqrt{r}$$

LP portfolio value (in terms of token B):

$$V_{\text{LP}} = x_1 \cdot P_1 + y_1 = 2y_0\sqrt{r}$$

HODL portfolio value (if LP had just held the tokens):

$$V_{\text{HODL}} = x_0 \cdot P_1 + y_0 = y_0(r+1)$$

Impermanent loss:

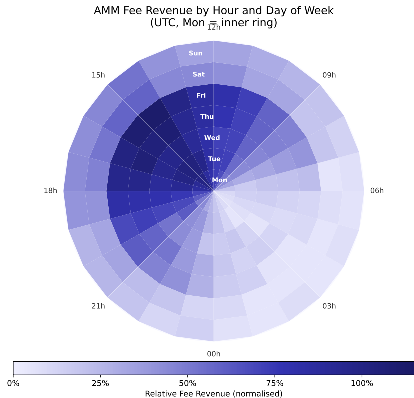
$$\boxed{\text{IL}(r) = \frac{V_{\text{LP}}}{V_{\text{HODL}}} - 1 = \frac{2\sqrt{r}}{r+1} - 1}$$

Numerical examples: $r = 1.25 \Rightarrow \text{IL} = -0.6\%$, $\quad r = 2 \Rightarrow \text{IL} = -5.7\%$, $\quad r = 5 \Rightarrow \text{IL} = -25.5\%$.

Impermanent loss is permanent unless the price returns to the entry ratio. At 2x price change, LPs lose 5.7 percent vs holding. Most Uniswap v2 LPs lose money after accounting for IL – fees rarely compensate.

(c) Joerg Osterrieder 2026

When Do AMMs Generate the Most Revenue – and Who Captures It?



Fee revenue patterns:

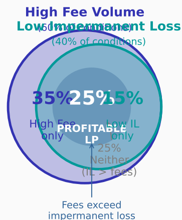
- **Peak hours:** 14:00–18:00 UTC (US market open overlaps with European afternoon)
- **Peak days:** Tuesday–Thursday concentrate highest volume
- **Dead zones:** weekend nights (00:00–06:00 UTC Saturday/Sunday)
- **Revenue split:** LPs receive fees minus MEV extraction

The circular heatmap reveals both hourly and weekly cycles. Active LPs who concentrate liquidity during peak periods earn disproportionately higher returns.

AMM fee revenue concentrates in weekday peak hours (14:00–18:00 UTC). The circular heatmap reveals both hourly and weekly patterns – LPs who optimize timing can significantly improve returns.

When Do Liquidity Providers Actually Make Money?

When Do Liquidity Providers Actually Make Money?



Proportional Euler diagram | Illustrative: share of market conditions 2021–2024

The LP profitability Venn:

- **Circle 1 – High fees:** volatile pairs with high volume generate the most fee income
- **Circle 2 – Low IL:** stable pairs with mean-reverting prices minimize impermanent loss
- **Overlap:** the sweet spot is narrow – high volume but mean-reverting prices (e.g., correlated stablecoin pairs)

Empirical finding: roughly 40% of Uniswap v3 positions are net profitable. The rest lose to IL after fees.

The conditions that generate high fees (volatility) are the same conditions that generate high impermanent loss. The profitable overlap is smaller than most LPs realize – roughly 40 percent of Uniswap v3 positions are net profitable.

(c) Joerg Osterrieder 2026

Can You Simulate a Constant Product AMM and Calculate Impermanent Loss?

```python
import numpy as np
def swap(x, y, dx, fee=0.003):
    dx_eff = dx * (1 - fee)
    dy = y * dx_eff / (x + dx_eff)
    return x + dx, y - dy, dy
def impermanent_loss(r):
    return 2 * np.sqrt(r) / (r + 1) - 1

# Simulate 1000 random trades on ETH/USDC pool
x, y, fees = 100.0, 300_000.0, 0.0
for _ in range(1000):
    dx = np.random.uniform(-2, 2)
    if dx > 0:
        x, y, dy = swap(x, y, dx)
        fees += dx * 0.003 * y / x
    elif dx < 0:
        y, x, _ = swap(y, x, -dx * y / x)
print(f"IL: {impermanent_loss((y/x)/3000):.4%} Fees: ${fees:.0f}")
```

Run this simulation with different volatility levels. High volatility generates more fee income but also more impermanent loss. The question is which effect dominates – and the answer depends on the specific pool and time period.

What Is the Formal Profit Condition for a Flash Loan Attack?

Flash loan constraint. Borrow amount B, fee ϕ (typically 0.09%). The transaction must repay $B + \phi B$ atomically or the entire transaction reverts.

Profit condition. Let $f(B)$ be the return from deploying borrowed capital:

$$\boxed{\pi = f(B) - B - \phi B \geq 0 \quad \Longleftrightarrow \quad f(B) \geq B(1+\phi)}$$

Arbitrage profit. Price discrepancy ΔP between venues A and B on amount B:

$$\pi_{\text{arb}} = B \cdot \Delta P - \phi B - G_{\text{gas}} \cdot P_{\text{gas}}$$

Manipulation profit. Attacker manipulates a price oracle, borrows against inflated collateral:

$$\pi_{\text{manip}} = \text{LTV} \cdot V_{\text{inflated}} - B(1+\phi) - G_{\text{gas}} \cdot P_{\text{gas}}$$

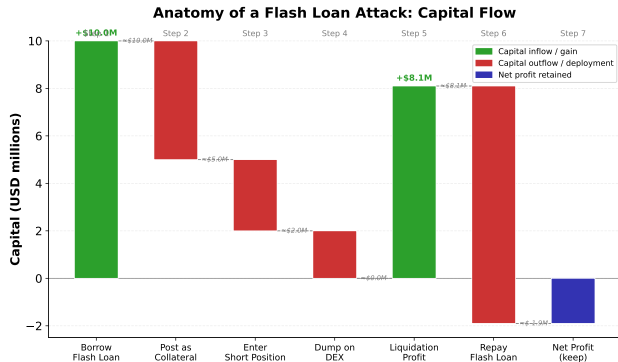
Risk profile. Flash loans create an *asymmetric* payoff:

$$\mathbb{E}[\pi] = p_{\text{success}} \cdot \pi_{\text{gross}} - (1 - p_{\text{success}}) \cdot G_{\text{gas}} \cdot P_{\text{gas}}$$

Failure cost is only the gas fee (∼\$10). This makes attacks rational even at low p_{success}.

Flash loans create an asymmetric risk profile: unlimited upside if the attack succeeds, \$10 gas cost if it fails. This is why flash loan attacks are rational even at low success probabilities.

What Does a Flash Loan Attack Look Like from the Inside?



Anatomy of a flash loan attack:

- **Step 1 – Borrow:** take unlimited loan from Aave/dYdX (fee ∼0.09%)
- **Step 2 – Manipulate:** crash or inflate a price on a low-liquidity pool
- **Step 3 – Exploit:** use manipulated price to extract value from a dependent protocol
- **Step 4 – Repay:** return loan + fee atomically

Key insight: every step happens in *one transaction*. If any step fails, the entire sequence reverts – zero capital at risk.

Every flash loan attack is a waterfall of capital flowing through composable protocols. The entire sequence – borrow, manipulate, profit, repay – happens atomically in one block.

What Is the Nash Equilibrium of the Sandwich Attack Game?

Setup. A victim submits a swap of size Δx_v on an AMM with reserves (x, y) and slippage tolerance s.
Bot strategy. The sandwich bot inserts a frontrun of size Δx_f before and a backrun after the victim:
Bot profit (frontrun + victim trade + backrun):

$$\pi_{\text{bot}}(\Delta x_f) = \underbrace{P_{\text{after victim}} \cdot \text{tokens}_f}_{\text{backrun revenue}} - \underbrace{\Delta x_f}_{\text{frontrun cost}} - \underbrace{\text{priority fee}}_{\text{block builder bribe}}$$

Victim loss. Additional slippage from the frontrun:

$$L_{\text{victim}} = \Delta y_{\text{without sandwich}} - \Delta y_{\text{with sandwich}} \approx \frac{\Delta x_f \cdot \Delta x_v}{x^2} \cdot y$$

Optimal frontrun size. First-order condition $\partial \pi_{\text{bot}} / \partial \Delta x_f = 0$ yields:

$$\Delta x_f^* = \sqrt{x \cdot \Delta x_v} - x \qquad \text{(square root of pool size times victim trade)}$$

Nash equilibrium. With n competing bots, profit per bot $\to 0$ as $n \to \infty$. Surplus transfers to validators via priority fees. Retail traders always lose.

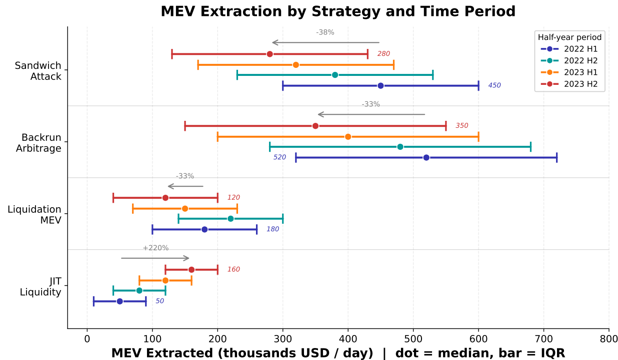
In the sandwich game's Nash equilibrium, bots compete away their profits as priority fees to validators. Retail traders always lose – the question is whether bots or validators capture the surplus.

Can You Simulate a Sandwich Attack and Calculate the Optimal Frontrun Size?

```python
import numpy as np
def sandwich_profit(x, y, victim_dx, front_dx, fee=0.003):
    """Calculate sandwich bot profit from frontrun + backrun."""
    # Frontrun: bot buys, price moves up
    eff = front_dx * (1 - fee)
    y1 = y - y * eff / (x + eff)
    x1 = x + front_dx
    # Victim trades at worse price
    eff_v = victim_dx * (1 - fee)
    x2, y2 = x1 + victim_dx, y1 - y1 * eff_v / (x1 + eff_v)
    # Backrun: bot sells tokens received from frontrun
    tokens = y - y1
    sell_eff = tokens * (1 - fee)
    dx_back = x2 * sell_eff / (y2 + sell_eff)
    return dx_back - front_dx # net profit in token A
x, y = 1000, 3_000_000 # ETH/USDC pool
for f in [0.5, 1, 2, 5, 10, 20]:
    print(f"Front={f:5.1f} ETH Profit={sandwich_profit(x,y,10,f):.2f}")
```

The optimal frontrun size balances profit (more frontrun = more slippage captured) against pool impact (too much frontrun = bot's own backrun gets worse price). Run this code to find the sweet spot.

How Much Value Do MEV Bots Extract – and Which Strategies Dominate?



MEV extraction landscape:

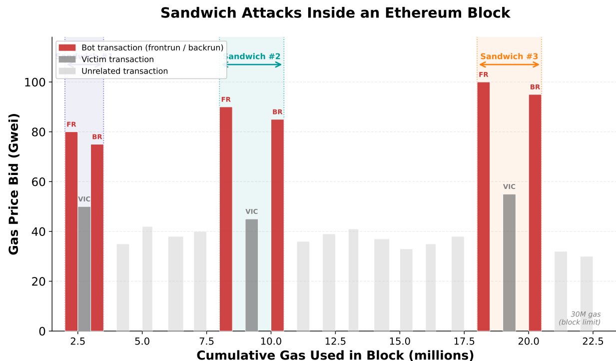
- **Sandwich attacks:** largest category, 60% of total MEV
- **Arbitrage:** price alignment across DEXes, 25%
- **Liquidations:** collateral seizure in lending protocols, 10%
- **Just-in-time liquidity:** concentrated LP for single blocks, 5%

Trend: Flashbots PBS redirected ∼40% of sandwich profits to validators in 2023. MEV is not disappearing – it is being redistributed up the supply chain.

MEV extraction is not decreasing – it is being redistributed. Flashbots redirected 40 percent of sandwich profits to validators in 2023. The invisible tax on DeFi users remains.

(c) Joerg Osterrieder 2026

What Does a Sandwich Attack Look Like Inside a Block?



Inside a single block:

- **Top layer:** block builder arranges transactions to maximize total MEV
- **Sandwich pair:** frontrun (bot buy) and backrun (bot sell) bracket the victim trade
- **Priority ordering:** bots bid for position via priority fees
- **Timing:** the entire sandwich executes in $\sim$12 seconds (one block)

The flame chart visualizes transaction ordering within a block – making the invisible MEV extraction visible. Each horizontal band is one transaction.

Inside a single Ethereum block, sandwich bots compete for transaction ordering. The flame chart makes visible the invisible: how block builders arrange transactions to maximize MEV extraction.

What Does It Cost to Buy a Vote in a Decentralized Democracy?

Governance voting power. A proposal passes when votes exceed quorum Q:

$$\sum_{i \in \text{supporters}} w_i \geq Q \qquad \text{where } w_i = \text{token balance of voter } i$$

Traditional attack cost. Buy Q tokens on the market:

$$C_{\text{buy}} = Q \cdot P_{\text{token}} + \text{slippage}(Q) \qquad \text{(millions of dollars for large protocols)}$$

Flash loan attack cost. Borrow Q tokens for one block:

$$C_{\text{flash}} = \phi \cdot Q \cdot P_{\text{token}} \qquad \text{where } \phi \approx 0.0009 \text{ (Aave fee)}$$

Cost reduction:

$$\frac{C_{\text{flash}}}{C_{\text{buy}}} = \phi \approx 0.09\% \qquad \Rightarrow \quad \sim 1{,}000 \times \text{ cheaper}$$

Defenses:

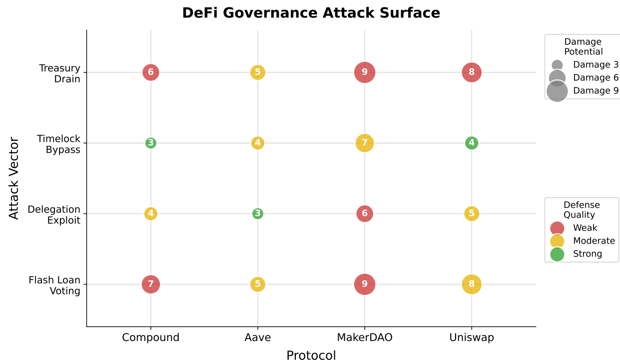
- **Timelock:** delay execution by τ blocks (typically 2–7 days) $\Rightarrow$ flash loan expires before vote executes
- **Snapshot voting:** use token balances from block $n - k$, not current block
- **Quorum + supermajority:** require $Q > 10\%$ of supply AND $> 66\%$ approval

---

A flash loan reduces governance attack cost by 1000x: from buying tokens (millions of dollars) to borrowing them (a 0.09 percent fee). Timelocks and quorums are the primary defenses.

(c) Joerg Osterrieder 2026

Which Protocols Are Most Vulnerable to Governance Attacks – and How?



Governance vulnerability factors:

- **Token concentration:** higher Gini = fewer holders needed for quorum
- **Timelock duration:** shorter = less time for community to react
- **Quorum threshold:** lower = easier to pass malicious proposals
- **Voter apathy:** typical turnout is 5–15% of eligible tokens

Case study: Compound Proposal 62 (2021) accidentally sent $80M to the wrong address. The timelock *prevented a fast fix* – governance could not correct its own mistake quickly enough.

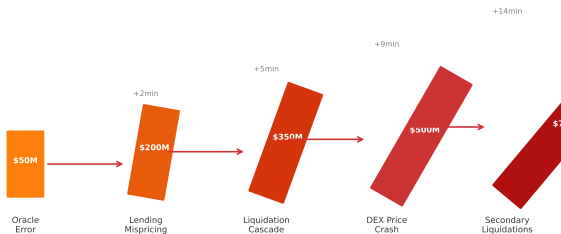
No DeFi protocol has solved governance security. The Compound Proposal 62 bug (2021) accidentally sent $80M to the wrong address – and governance could not fix it fast enough because of its own timelock.

Can You Detect When an Oracle Price Feed Is Being Manipulated?

```python
import numpy as np
def twap(prices, window=10):
    """Time-weighted average price over window blocks."""
    return np.mean(prices[-window:]) if len(prices) >= window else np.mean(prices)

def detect_manipulation(spot, twap_val, threshold=0.05):
    dev = abs(spot - twap_val) / twap_val
    return dev > threshold, dev

# Simulate: normal prices then attacker crashes price 30%
np.random.seed(42)
prices = [3000 + np.random.normal(0, 15) for _ in range(50)]
prices.append(3000 * 0.7) # manipulation in block 51

spot, tw = prices[-1], twap(prices, 10)
flagged, dev = detect_manipulation(spot, tw)
print(f"Spot=${spot:.0f} TWAP=${tw:.0f} Dev={dev:.1%} Flagged={flagged}")
# Spot=$2100 TWAP=$2994 Dev=29.9% Flagged=True
```

TWAP oracles resist single-block manipulation by averaging over N blocks. But they add latency – a 10-block TWAP on Ethereum means 2 minutes of stale data. Security and freshness trade off.

What Happens When One Corrupted Price Feed Cascades Through DeFi?



Oracle cascade anatomy:

- **Stage 1:** attacker manipulates a single price feed on a low-liquidity venue
- **Stage 2:** dependent lending protocols read the corrupted price as collateral value
- **Stage 3:** overleveraged borrowing against inflated collateral drains the lending pool
- **Stage 4:** liquidation cascades spread to other protocols sharing the same oracle

Case study: Mango Markets ($117M, 2022) – one corrupted price feed triggered overleveraged borrowing that drained the entire protocol in minutes.

The Mango Markets exploit ($117M, 2022) was a textbook oracle cascade: manipulated price feed led to overleveraged borrowing led to protocol drain. One corrupted data point, systemic failure.

(c) Joerg Osterrieder 2026

How Do You Quantify the Risk of Interconnected Protocols Failing Together?

Independent failure model. If each of n protocols has independent failure probability p:

$$P(\text{at least one failure}) = 1 - (1-p)^n$$

Correlated failure model. With pairwise correlation ρ between protocol failures:

$$P(A_i \cap A_j) = p^2 + \rho \cdot p(1-p) \qquad \text{for any pair } (i,j)$$

Cascade probability. Given one protocol has failed, the probability of a cascade to k others:

$$P(\text{cascade} \mid \text{one failure}) = 1 - (1-\rho)^{n-1}$$

Systemic risk measure. Expected number of cascading failures:

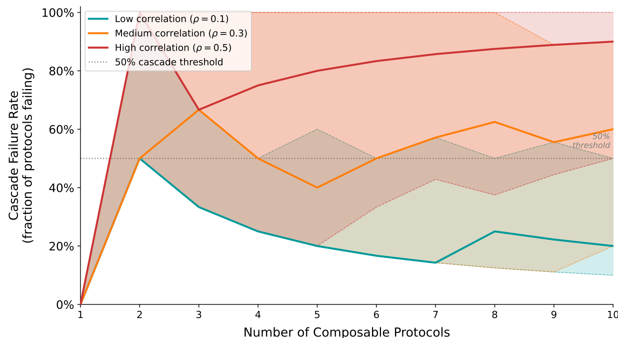
$$R_c = (n-1) \cdot \rho \cdot p + \binom{n-1}{2} \cdot \rho^2 \cdot p$$

Quadratic growth. Risk grows as $O(n^2\rho^2)$ – *not* linearly with the number of composable protocols.
Numerical example: $n = 5$, $p = 0.05$, $\rho = 0.3$:

$$P(\text{cascade} \mid \text{one failure}) = 1 - (1-0.3)^4 = 1 - 0.2401 = 76\%$$

Composability converts linear risk (independent failures) into quadratic risk (correlated cascades). Five composable protocols with 30 percent correlation have a 76 percent cascade probability when one fails.

How Fast Does Risk Grow as You Stack More Protocols Together?



The composability risk cone:

- **1–2 protocols:** manageable, independent risk dominates
- **3–4 protocols:** correlation effects emerge, cone begins to widen
- **5–6 protocols:** cascade probability exceeds 50% at moderate correlation
- **8+ protocols:** near-certain cascade within a year under realistic failure rates

Implication: the cone widens super-linearly. This is the mathematical case for circuit breakers – automated pause mechanisms that halt composable interactions when anomalies are detected.

The cone widens super-linearly: 3 composable protocols are manageable, 5 are risky, 8+ are near-certain to cascade within a year. This is the mathematical case for circuit breakers in composable DeFi.

Can You Simulate a Cascading Failure in a Composable Protocol Stack?

```python
import numpy as np
def simulate_cascade(n_proto, corr, p_fail, n_sims=10000):
    """Monte Carlo: how often does 1 failure cascade to k?"""
    cascades = []
    for _ in range(n_sims):
        failed = [np.random.random() < p_fail]
        for i in range(1, n_proto):
            if any(failed):
                p_c = 1 - (1 - corr) ** sum(failed)
                failed.append(np.random.random() < p_c)
            else:
                failed.append(np.random.random() < p_fail)
        cascades.append(sum(failed))
    return np.mean(cascades), np.percentile(cascades, 95)

for rho in [0.0, 0.1, 0.3, 0.5]:
    avg, p95 = simulate_cascade(5, rho, 0.02)
    print(f"rho={rho:.1f} Avg={avg:.2f} 95th={p95:.0f}/5")
```

At zero correlation, 5 protocols average 0.1 failures. At 0.3 correlation, the average jumps to 1.4 with 95th percentile at 4 out of 5. Composability converts independent risk into correlated catastrophe.

(c) Joerg Osterrieder 2026

What Have We Learned – and What Remains Unsolved in DeFi Engineering?

Section 1: Solidity Fundamentals. The EVM is a deterministic stack machine where storage costs 7,000x more than memory. EIP-1559 makes Ethereum deflationary under sustained demand. Gas optimization is fundamentally about minimizing state changes.

Section 2: Vulnerability Taxonomy. Over 80% of exploits fall into four families (state, arithmetic, access, external). Reentrancy is a graph reachability problem solvable in $O(V + E)$. No single verification tool catches all bug classes.

Section 3: AMM Mathematics. The constant product formula $xy = k$ creates a market from pure math. Impermanent loss is permanent unless prices revert. Only ~40% of LP positions are net profitable.

Section 4: Flash Loans and MEV. Flash loans create asymmetric risk (unlimited upside, $10 downside). Sandwich attacks converge to a Nash equilibrium where validators capture the surplus. MEV is a ~$500M/year invisible tax on DeFi users.

Section 5: Governance and Composability. Flash loans reduce governance attack costs by 1,000x. Oracle cascades convert single failures into systemic crises. Composability risk grows quadratically with the number of stacked protocols.

What remains unsolved: cross-protocol verification, MEV-resistant AMM designs, governance mechanisms resistant to flash loan attacks, and circuit breakers that preserve composability while limiting cascades.

Five sections, one paradox: permissionless composability creates both the innovation and the attack surface. The open question is whether DeFi can keep the composability while adding the circuit breakers.

What Are the Six Engineering Lessons Every DeFi Developer Must Internalize?

1. **Storage is expensive:** every SSTORE is a 20,000-gas tax. Cache in memory, batch writes, minimize state changes. Gas optimization is not premature – it is the cost of decentralization.
2. **Reentrancy is a graph problem:** CEI (Checks-Effects-Interactions) is a topological constraint on the call graph. If external calls precede state writes in any cycle, the contract is vulnerable.
3. **Impermanent loss is usually permanent:** at 2x price change, LPs lose 5.7% vs holding. Most LP positions lose money. Fees must exceed IL for profitability – and they usually do not.
4. **Flash loans break all capital assumptions:** any attacker has unlimited capital for one transaction at 0.09% cost. Design protocols assuming every user has infinite funds.
5. **MEV is an invisible tax:** sandwich attacks, arbitrage, and liquidations extract ~$500M/year from DeFi users. Transaction ordering is a first-class security concern.
6. **Composability risk is quadratic:** stacking n protocols creates $O(n^2)$ correlated failure modes. The composability that makes DeFi innovative also makes it fragile.

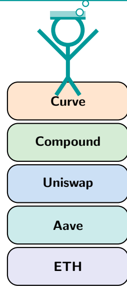
Six takeaways, one framework: DeFi composability is a double-edged sword. The same permissionless openness that enables money legos enables money grenades. Engineering the boundary between innovation and fragility is the central challenge.

References and Further Reading

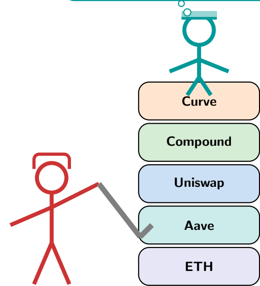
1. Daian, P., Goldfeder, S., Kell, T., et al. (2020). Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. *IEEE S&P 2020*.
2. Adams, H., Zinsmeister, N., Salem, M., Keefer, R., Robinson, D. (2021). Uniswap v3 Core. *Uniswap Whitepaper*.
3. Qin, K., Zhou, L., Gervais, A. (2022). Quantifying Blockchain Extractable Value: How Dark is the Forest? *IEEE S&P 2022*.
4. Perez, D., Werner, S., Xu, J., Livshits, B. (2021). Liquidations: DeFi on a Knife-Edge. *Financial Cryptography 2021*.
5. Zhou, L., Qin, K., Cully, A., Livshits, B., Gervais, A. (2023). SoK: Decentralized Finance (DeFi) Attacks. *IEEE S&P 2023*.
6. Gudgeon, L., Perez, D., Harz, D., Livshits, B., Gervais, A. (2020). The Decentralized Financial Crisis. *Crypto Valley Conference on Blockchain Technology*.

Start with Daian et al. (2020) for MEV, Adams et al. (2021) for AMM math, and Zhou et al. (2023) for the definitive attack taxonomy.

"I understand money legos now!"



"...I also understand why they break."



The composability paradox: you cannot understand DeFi innovation without also understanding DeFi fragility.

(c) Joerg Osterrieder 2026