Smart Contracts

Learning Outcome: Smart Contracts analysieren und verstehen — Bloom's: Analyze

Prof. Dr. J. Osterrieder

BSc Blockchain, Crypto Economy & NFTs

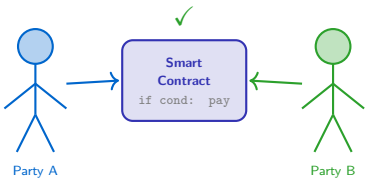
Spring 2026

What If a Contract Could Enforce Itself?

Traditional contracts depend on:

- Courts and legal systems to adjudicate disputes
- Lawyers and intermediaries to draft and verify terms
- Trust that each party will fulfill obligations
- Enforcement mechanisms that are slow and costly

Smart contracts change this: Code deployed on a blockchain executes automatically when conditions are met—no courts, no lawyers, no counterparty trust required.



Key insight: Smart contracts replace trust in people with trust in code—verifiable, deterministic, and censorship-resistant.

Anatomy of a Smart Contract

Definition

A smart contract is code deployed on a blockchain that **holds state**, **accepts transactions**, and **executes logic automatically** without any central authority.

Four core components:

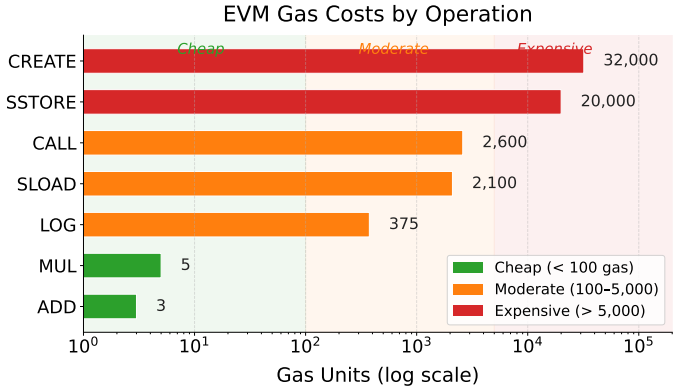
1. **State variables** — persistent data stored on-chain
2. **Functions** — callable logic that reads or modifies state
3. **Events** — logs emitted to the blockchain for off-chain listeners
4. **Modifiers** — reusable access control and precondition checks

What each component does:

- **State variables:** Balances, ownership records, flags—anything that must persist between calls
- **Functions:** The verbs of a contract (`transfer`, `approve`, `mint`)—can be public, private, or restricted
- **Events:** Enable DApps to react to contract activity without polling the chain
- **Modifiers:** Enforce rules like `onlyOwner` or `whenNotPaused` before a function runs

Analyze a contract by reading these four layers in order: what it stores, what it does, what it signals, and who can call it.

What Does Execution Actually Cost?



- **Gas** = unit of computational work on the EVM; every opcode has a fixed gas cost defined in the Yellow Paper
- **Transaction fee** = gas price (Gwei, set by user) $\times$ gas used (determined by code path)
- **Storage dominates:** `SSTORE` costs 20,000 gas vs. `ADD` at 3 gas—a 6,000$\times$ difference

Implication: Minimizing storage writes is the single most impactful optimization available to smart contract developers.

Reading a Smart Contract

ERC-20 `transfer` function: `function transfer(address to,`

```
uint256 amount) {
  require(
    balance[msg.sender] >= amount
  );
  balance[msg.sender] -= amount;
  balance[to] += amount;
  emit Transfer(
    msg.sender, to, amount
  );
}
```

Line-by-line analysis:

- `require(...)` — **Guard clause:** reverts the entire transaction if sender has insufficient balance; no partial execution
- `balance[msg.sender] -= amount` — **Debit:** reduces sender's stored balance; costs one `SSTORE`
- `balance[to] += amount` — **Credit:** increases recipient's balance; costs a second `SSTORE`
- `emit Transfer(...)` — **Event log:** records transfer for wallets and block explorers; cheaper than storage

Pattern: Every well-written function follows CHECK – EFFECT – INTERACT. Violations of this order create reentrancy vulnerabilities.

Worked Example: Token Transfer Cost

Gas breakdown for one ERC-20 `transfer` call:

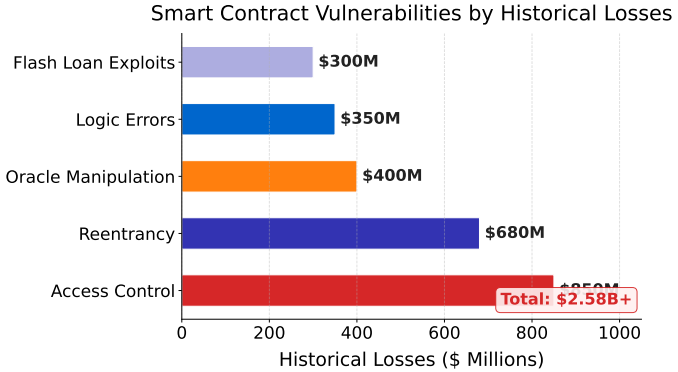
Operation	Opcode	Gas cost
Base transaction fee	—	21,000
Load sender balance	`SLOAD`	2,100
Write sender balance	`SSTORE`	5,000
Write receiver balance	`SSTORE`	5,000
Emit Transfer event	`LOG3`	375
Total		**33,475**

Total transaction cost:

$$\underbrace{33{,}475 \text{ gas}}_{\text{computation}} \times \underbrace{30 \text{ Gwei}}_{\text{gas price}} = \underbrace{1{,}004{,}250 \text{ Gwei}}_{\text{total cost}} = \underbrace{0.001 \text{ ETH}}_{\approx \$3 \text{ at } \$3{,}000/\text{ETH}}$$

At peak congestion (300+ Gwei), this same transfer costs $30+. Gas awareness is essential for DApp usability.

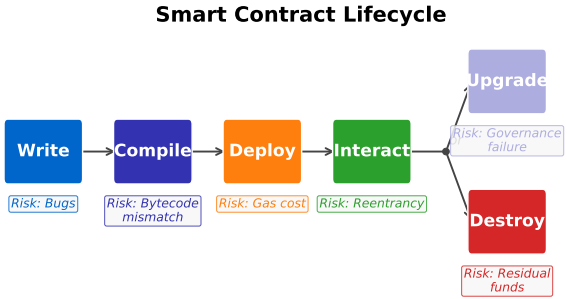
Where Smart Contracts Fail



- **Reentrancy:** Attacker re-enters before state updates; exploited in The DAO hack (2016, $60M), triggering Ethereum's hard fork
- **Access control failures:** Missing `onlyOwner` checks are the most common vulnerability by total USD lost in DeFi
- **Oracle manipulation:** Flash loans distort on-chain price feeds within one transaction, draining lending reserves

Audit checklist: reentrancy guards, proper access modifiers, and time-weighted average prices (TWAP) for oracle inputs.

The Smart Contract Lifecycle



Immutability (default):

- Deployed bytecode cannot be changed
- Address fixed; state persists on-chain
- No backdoor edits possible
- Bugs are permanent without migration

Upgradeability (proxy patterns):

- Proxy holds state; delegates to implementation
- Owner re-points proxy to new logic contract
- User balances preserved across upgrades
- Trade-off: trust in the upgrader/governance

Most major DeFi protocols use transparent or UUPS proxy patterns—upgradeability is a feature, not a flaw, when governance controls it.

Four Questions to Analyze Any Contract

The four analytical questions:

1. **What state does it hold?**
   Inventory all storage variables and their types
2. **What can go wrong?**
   Screen for known vulnerability classes
3. **Who can call it?**
   Map each function to its access level
4. **Is it upgradeable?**
   Identify proxy patterns and owner privileges

How to answer each question:

- **Storage layout:** Read the state variable declarations at the top of the contract; note mappings, arrays, and structs
- **Vulnerability scan:** Check for `call.value` before state update (reentrancy), missing `require` on sensitive paths, and external price feed usage
- **Access modifiers:** Enumerate `public`, `external`, `onlyOwner`, and `internal` on every function
- **Proxy detection:** Search for `delegatecall`, `implementation` slot, or `upgradeTo` function in the ABI

Applying these four questions to any contract reduces audit time and ensures no critical attack surface is overlooked.

Key Takeaways

What you have learned:

1. Smart contracts are **self-executing programs** on a blockchain—they replace institutional trust with code logic
2. Contracts consist of **state, functions, events, and modifiers**—each layer has a distinct analytical role
3. **Gas cost** is dominated by storage operations; `SSTORE` is orders of magnitude more expensive than arithmetic
4. **Vulnerability classes**—reentrancy, access control, oracle manipulation—follow predictable patterns that auditors scan systematically
5. **Immutability vs. upgradeability** is a design choice: proxies enable bug fixes but introduce governance trust assumptions

You can now analyze:

- **Contract structure:** Read state variables, identify functions, trace modifier chains, and interpret event logs
- **Gas costs:** Decompose any transaction into opcode-level costs and estimate fees at a given gas price
- **Vulnerability types:** Recognize reentrancy, access control gaps, and oracle attack surfaces in source code
- **Lifecycle stages:** Distinguish immutable vs. upgradeable deployments and assess the governance implications of each

Bloom's **Analyze**: decompose, differentiate, examine, and attribute—applied to smart contract code.