Access Control in Ethereum Smart Contracts

L. Olivieri D. Ressi A. Spanò

DLT Workshop, June 2026

- Ethereum is a **permissionless** execution environment
- Any account can invoke public functions of a smart contract
- **authorization logic must be implemented manually**

- Access control failures caused:
  - Parity Wallet exploits
  - ownership hijacking
  - unrestricted withdrawals
  - delegatecall abuses
- In 2024 alone:

$$> \$953.2 \text{ million lost}$$

due to access-control-related vulnerabilities

Example 1: DAC

A simple mechanism to protect a **privileged** function.

```
contract C {
  address private owner;

  constructor() {
    // the owner is who constructs this contract
    owner = msg.sender;
  }

  // only the owner can invoke this function
  function priliveged() public {
    require(msg.sender == owner, "Not authorized");
    // perform some privileged operation
  }
}
```

Control logic is totally **up to the programmer**.

How Contrived is This?

A parallel with ordinary operating systems:

- Like if Unix/Linux did not have user/group permissions
- So that
  - every program
  - every command
  - every executable
- **had to manually check who's launching it!**

Example 2: RBAC

For **role-based** forms of AC, things get even more *complicated*.

```
import "@openzeppelin/contracts/access/AccessControl.sol";

contract MyRBAC is AccessControl { // inherits superclass

  // roles are just (hashed) strings
  bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");
  bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");

  // only who has the minter role can call
  function mint(address to, uint256 amt) public onlyRole(
      MINTER_ROLE) { ... }

  // only admin can assign the minter role to a new user
  function addMinter(address a) public onlyRole(ADMIN_ROLE) { ... }
}
```

Libraries exist to support programmers.

Motivation

The bottom line is:

- whatever mechanism you need
  - *everything is up to you!*
  - no support from the underlying platform
  - or from the language

This implies a **high error-proneness**.

We want to investigate the state of AC in smart contracts.

Research Questions and Contributions

RQ1 How is access control implemented in Ethereum smart contracts?

RQ2 To what extent are standardized implementations adopted?

RQ3 What recurring coding patterns lead to access control vulnerabilities?

RQ4 How do existing tools detect access control vulnerabilities?

Main contributions

- empirical study over 181,717 deployed contracts (**RQ1-2**)
- structured taxonomy of AC vulnerabilities (**RQ3**)
- evaluation of existing analyzers (**RQ4**)

Contribution 1

Empirical Study

Empirical Study

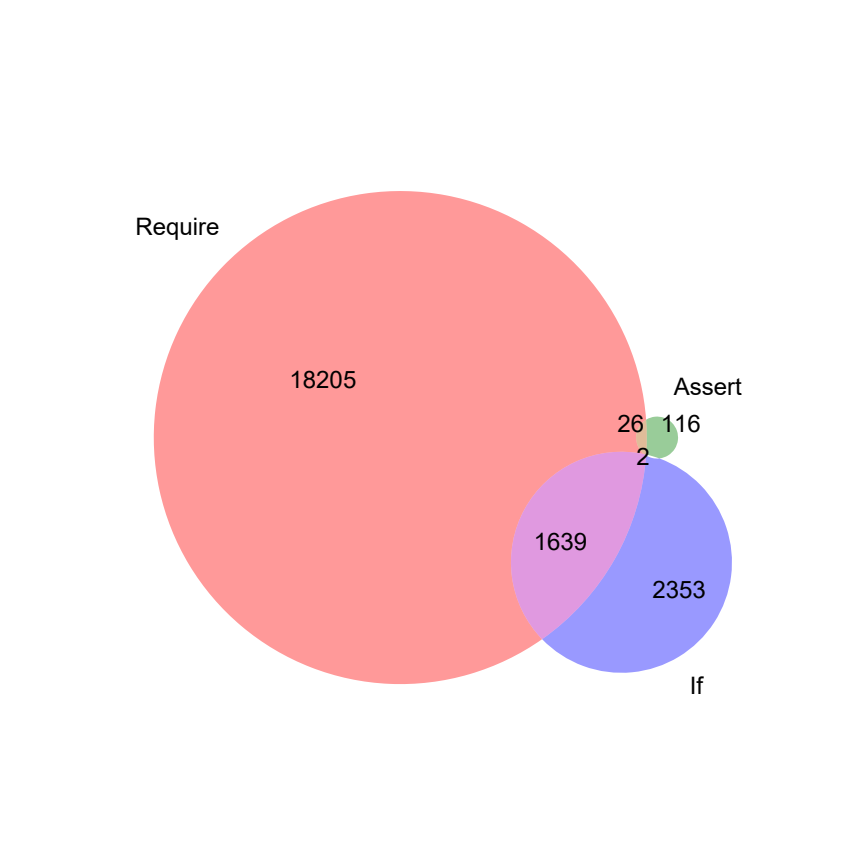
Dataset

- Zellic 2023 Smart Contract Source Index [1]
- Ethereum mainnet contracts
- Deduplicated using SHA256
- 181,717 unique Solidity files

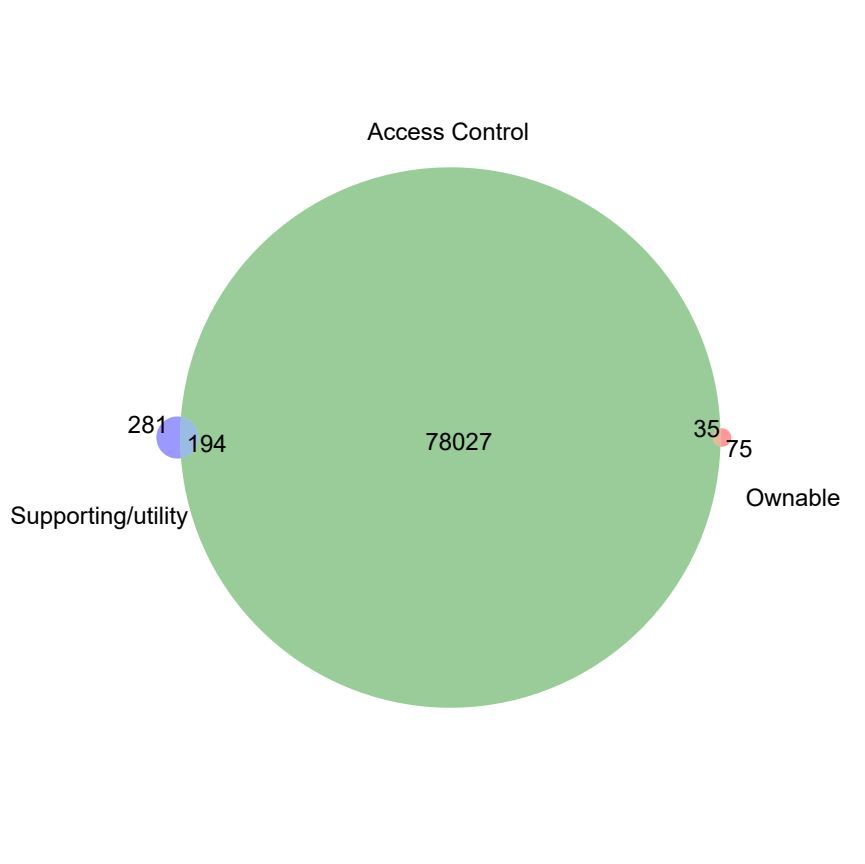
Methodology

- regex-based large-scale analysis
- detection of:
  - `msg.sender` checks
  - OpenZeppelin imports [2]
- scalable across Solidity versions

Empirical Study



(a) Manual checks based on `msg.sender`



(b) Using OpenZeppelin imports

Empirical Study

Finding 1

Only 12.29% of contracts explicitly implement sender-based checks.

Finding 2

About 43.3% import OpenZeppelin access control patterns.

- many contracts rely on reusable abstractions
- interfaces are imported more often than implementations

Implication
AC is widespread and non-standard $\Rightarrow$ hard to detect syntactically.

Contribution 2

A Novel Taxonomy

A Novel Taxonomy

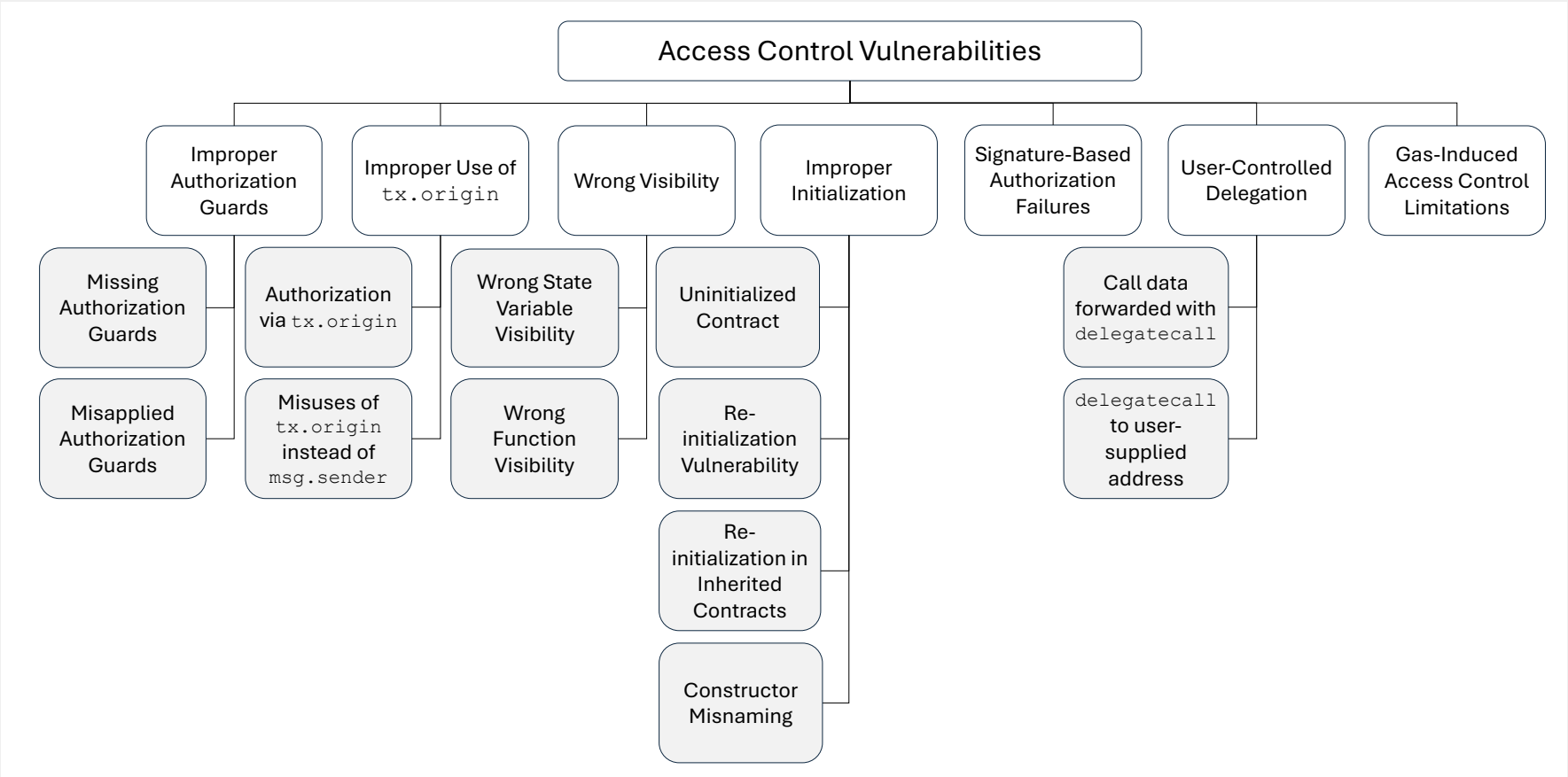
Existing taxonomies are:

- coarse-grained
- overlapping
- often syntactically biased

We propose a taxonomy organized as a **tree-like hierarchy**:

- granular systematization
- semantic-based
- with mappings to:
  - Common Weaknesses Enumeration (CWE) [3, 4]
  - Smart Contract Weakness Classification (SWC) [5]
  - Taxonomy of AC Vulnerabilities (TACV) [6],
  - OpenSCV [7]
  - Consolidated Taxonomy of Vulnerabilities (CTV) [8]

A Novel Taxonomy



We handcrafted a dataset of MWEs belonging to these categories.

Contribution 3

Tool Evaluation

How Tools Perform on our Dataset?

Appendix	Description	CCC	Mando	Mythril	Slither	Smartcheck	Solhint
B.1.1	Missing authorization guards	✓	✓				
B.1.2	Incorrect logical condition		✓				
B.1.2	Missing revert in conditional branch		✓				
B.2.1	Authorization via `tx.origin`	✓	✓	✓	✓	✓	✓
B.2.2	Misuses of `tx.origin` instead of `msg.sender`	✓	✓			✓	✓
B.3.1	Wrong state variable visibility		✓				
B.3.2	Wrong function visibility		✓				
B.4.1	Uninitialized contract		✓				✓
B.4.2	Re-initialization vulnerability		✓				
B.4.3	Re-initialization in inherited contracts		✓				
B.4.4	Constructor misnaming					✓	
B.5	Signature-based authorization failures			✓			
B.6.1	Call data forwarded with `delegatecall`	✓	✓	✓	✓		
B.6.2	`delegatecall` to user supplied address		✓	✓	✓		
B.7	Gas-induced access control limitations	✓				✓	✓

Tool Evaluation

Evaluated via SmartBugs

Tool	Coverage Trend
Slither	narrow rules
Mythril	semantic tx.origin checks
SmartCheck	syntactic patterns
Mando	highest coverage
LLMs	strongest generalization

Most tools *detect* **narrow syntactic patterns**
And *fail* on **semantic variants**

LLMs vs Traditional Tools

Experiment

Yes, we also asked ChatGPT 5.3 to evaluate our handcrafted MWEs.

Result

- all vulnerabilities correctly identified
- additional issues often detected
- stronger syntactic generalization

Interpretation

LLMs appear better at:

- semantic reasoning
- intent inference
- pattern generalization

But:

- lack of formal guarantees
- difficult reproducibility

Future Directions

We might develop the ultimate AC static analyzer:

- using top-notch FM techniques
- abstract interpretation
- polyhedral domains

No easy task.

And what about LLMs? They perform well, though not perfectly.

Future Directions

Thank you for your attention.

References I

- Zellic, "Zellic 2023 Smart Contract Source Index," 2023, https://huggingface.co/datasets/Zellic/smart-contract-fiesta (Accessed 01/2026).
- OpenZeppelin, "OpenZeppelin Contracts Repository," 2026, https://github.com/OpenZeppelin/openzeppelin-contracts Accessed 03/2026.
- MITRE Corporation, "Common weakness enumeration (cwe)," https://cwe.mitre.org/, 2024, accessed: 2026-03-13.
- P. Mell and A. Gueye, "A suite of metrics for calculating the most significant security relevant software flaw types," in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2020, pp. 511–516.
- SWC Foundation, "Swc registry: Smart contract weakness classification," https://swcregistry.io/, 2023, accessed: 2026-03-11.

References II


H. Liu, D. Wu, Y. Sun, S. Wang, and Y. Liu, "Have we solved access control vulnerability detection in smart contracts? A benchmark study," in *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*. IEEE, 2025, pp. 1995–2007. [Online]. Available: https://doi.org/10.1109/ASE63991.2025.00166


F. R. Vidal, N. Ivaki, and N. Laranjeiro, "Openscv: an open hierarchical taxonomy for smart contract vulnerabilities," *Empirical Software Engineering*, vol. 29, no. 4, p. 101, 2024.


H. Rameder, M. Di Angelo, and G. Salzer, "Review of automated vulnerability analysis of smart contracts on ethereum," *Frontiers in Blockchain*, vol. 5, p. 814977, 2022.