

ATTACK-CENTRIC BY DESIGN: A PROGRAM-STRUCTURE TAXONOMY OF SMART CONTRACT VULNERABILITIES

PARSA HEDAYATNIA ¹, TINA TAVAKKOLI ¹, HADI AMINI ¹,
MOHAMMAD ALLAHBAKHSH ^{1,*}, HALEH AMINTOOSI ¹

¹Computer Engineering Department, Faculty of Engineering, Ferdowsi University of Mashhad (FUM), Mashhad, Iran

ABSTRACT. Smart contracts concentrate high-value assets and complex logic in small, immutable programs, where even minor bugs can cause major losses. Existing taxonomies and tools remain fragmented—organized around symptoms such as reentrancy rather than structural causes. This paper introduces an *attack-centric, program-structure* taxonomy that unifies Solidity vulnerabilities into eight root-cause families covering control flow, external calls, state integrity, arithmetic safety, environmental dependencies, access control, input validation, and cross-domain protocol assumptions. Each family is illustrated through concise Solidity examples, exploit mechanics, and mitigations, and linked to the detection signals observable by static, dynamic, and learning-based tools. We further cross-map legacy datasets (SmartBugs, SolidiFI) to this taxonomy to reveal label drift and coverage gaps. The taxonomy provides a consistent vocabulary and practical checklist that enable more interpretable detection, reproducible audits, and structured security education for both researchers and practitioners.

Keywords— Smart Contracts, Blockchain Security, Vulnerability Taxonomy, Solidity Security, Program Structure, Attack Surface, Root-Cause Analysis, Attack-Centric Classification.

1 Introduction

Smart contracts encode financial logic and governance rules that execute autonomously on blockchains. Their immutability, composability, and ubiquitous deployment elevate the security impact of defects: a single vulnerability can be replicated across thousands of instances, exploited permissionlessly, and amplified through inter-contract interactions. Over the last decade, the community has catalogued numerous failure modes and high-profile exploits, beginning with The Decentralized autonomous organization (DAO) and continuing through reentrancy variants, randomness manipulation, access-control bypasses, and proxy/upgradeability pitfalls. Early systematization by Atzei et al. [1] established a layered view (Solidity, Ethereum Virtual Machine (EVM), blockchain) that has since been refined by broad surveys and mappings [2–7].

This article is a *review focused on Solidity vulnerabilities*. Rather than centering on any single detection technique, we provide a consolidated, program-structure-oriented taxonomy of attack

E-mail address: parsa.hedayatnia@alumni.um.ac.ir, tina.tavakkoli@mail.um.ac.ir, hadi.amini@mail.um.ac.ir,
Date: November 13, 2025.

*Corresponding author: allahbakhsh@um.ac.ir.
*MSC2020: Primary 00A05, Secondary 00A66.

surfaces and show how concrete exploits arise from (i) control-flow interaction (e.g., reentrancy, transaction ordering dependence), (ii) external calls and exception handling, (iii) state/storage integrity (e.g., unsafe `delegatecall`), (iv) arithmetic and type safety, (v) blockchain-environment dependency (e.g., timestamp and on-chain randomness), (vi) access control/authentication (e.g., `tx.origin` misuse), (vii) Application Binary Interface (ABI)/input validation (e.g., short-address), and (viii) network/protocol issues. For each family, we relate historical exploits to formal root causes (data flow, control flow, storage aliasing, environment assumptions), bridging legacy names with modern Smart Contract Weakness Classification (SWC)-style categories and recent taxonomies.

Three trends justify a fresh, attack-centric consolidation. *First*, detection and assurance techniques have diversified: precise Control Flow Graph (CFG) recovery for bytecode [8], expert-rule/static analyzers [9, 10], dynamic/fuzzing ecosystems and injected benchmarks [11, 12], and multiple deep-learning lines—Abstract Syntax Tree (AST)/CFG/Graph neural network (GNN) and transformer models [13–20]. *Second*, the attack surface has shifted with upgradeable proxies, cross-contract patterns, and “long” contracts, pushing research toward multimodal fusion and efficiency [21–24]. *Third*, large language models (LLMs) and LLM-enhanced frameworks are being actively explored for triage, explanation, and hybrid pipelines [25–27].

Our contributions are as follows:

- A unified, attack-centric taxonomy aligned to program-structure root causes, reconciling legacy names with modern categories and mitigations [1–5, 7].
- A detector-signal view for each family (what static, dynamic, and learning-based tools must observe) and code-backed exemplars [8, 11, 12, 14–16, 25–28].
- A cross-mapping of benchmark labels (SmartBugs, SolidiFI, and derivatives) to reduce ambiguity and surface coverage gaps and sources of label drift [7, 11, 12].
- We highlight open challenges: (i) precise inter-contract reasoning at bytecode level [8], (ii) explainability and false-positive control for AI models [3, 6, 29], and (iii) scalable formal verification for real-world patterns (loops, calls, upgradeability) [28, 30, 31].
- All taxonomies, code listings, and mappings are designed for reuse in benchmarking, education, and the development of future detection tools.

While numerous works focus on *how* to detect vulnerabilities—GNNs with expert priors [32], hierarchical attention [33], CFG+Transformer [15], multimodal/entropy fusion [17, 18], BERT variants and semi-supervision [34], and practical tool integrations [10, 22]—our review centers on *what gets attacked and why*. We reference detection/verification literature only to ground each attack family in concrete mitigations and analysis affordances.

Paper organization.

Section 2 reviews related taxonomies, vulnerability detectors, and benchmark datasets. Section 3 introduces our unified vulnerability taxonomy grounded in eight program-structure root causes. Section 4 presents illustrative vulnerability examples with annotated code, exploitation mechanics, and mitigation guidance. Section 5 discusses implications for benchmarking, explainability, and dataset relabeling. Section 6 maps legacy/historical attack terminology to the unified taxonomy for consistent reporting. Section 7 concludes and outlines future work.

2 Related literature

Understanding and detecting smart contract vulnerabilities has spurred a rich and growing body of research, spanning taxonomic surveys, detection tools, evaluation benchmarks, and formal verification approaches. This section synthesizes the most influential studies across these themes to contextualize our taxonomy and highlight unresolved challenges. Whereas prior work

often organizes vulnerabilities by language features, detection heuristics, or incident outcomes, our contribution is distinguished by an attack-centric taxonomy rooted in program-structure dimensions. This structure helps reconcile historical naming inconsistencies and aligns detection signals, tool coverage, and dataset design under a unified analytical framework. We group the related literature into six key strands: foundational taxonomies, evaluation benchmarks, program analysis, machine learning, environment-based attacks, and LLM-integrated pipelines.

Foundational taxonomies and surveys. The SoK by Atzei et al. [1] introduced a layered taxonomy that remains a reference point for classifying Solidity/EVM/blockchain-level flaws. Recent surveys expand both breadth and depth: Chu et al. [2] organize vulnerabilities alongside data sources and repair; Vidal et al. [3] provide a systematic literature review emphasizing technique coverage and evaluation gaps; Jiao et al. [4] synthesize attacks and detection for Ethereum specifically; Kezadri Hamiaz & Driss [5] catalog tools and challenges across verification techniques; Zhu et al. [6] compare formal, fuzzing, Machine Learning (ML), and program-analysis pipelines; and Iuliano & Di Nucci [7] review vulnerabilities, tools, and benchmarks up to 2024, noting deficiencies in standard datasets and labeling.

Datasets, benchmarks, and evaluation. SmartBugs [11] curated real and synthetic vulnerable contracts with a uniform runner for tool evaluation; SolidiFI [12] introduced systematic bug injection for controlled benchmarking. Subsequent works reuse or extend these corpora (e.g., [17, 18, 22]) while highlighting threats to validity such as label noise and version skew [7]. Precise bytecode-level CFG reconstruction [8] improves static analyses and reduces spurious alarms for flow-sensitive categories (reentrancy, call-before-effects).

Static/dynamic program analysis. Representative static analyzers and hybrids span expert rules and symbolic reasoning [9, 10], with ongoing interest in exception handling, external calls, and proxy/upgrade patterns. Dynamic techniques and fuzzing ecosystems are commonly evaluated on SmartBugs/SolidiFI runners [11, 12]. Formal verification advances include modular reasoning and invariant synthesis for realistic Solidity patterns [28], surveys of verification approaches [30], and cross-layer verification challenges [31].

Learning-based detection. Deep learning directions capture syntax, structure, and semantics through AST/CFG graphs, token sequences, and opcode views: CNN/RNN hybrids [19, 35], GNNs with expert priors [14, 32, 33], transformer and attention-based models [15, 16], and multi-modal/entropy or decision fusion [13, 17, 18, 21, 22]. Semi-supervised and active-learning BERT variants [34] address label scarcity; recent work targets long contracts and few-shot regimes via feature fusion [23, 24]. Surveys focused on Deep Learning tooling report strengths and common failure modes [29].

Environment-driven attacks. Randomness misuse and miner influence on timestamps/blocks have been systematically analyzed by Qian et al. [36], providing a taxonomy and detectors that complement our environment-dependency family. Access-control pitfalls such as `tx.origin` misuse and suicidal contracts persist in both rule-based and learning datasets [2, 22, 35].

LLMs and hybrid pipelines. Recent studies discuss where LLMs help and where they fail in vulnerability detection: position papers and evaluations [25, 26] and LLM-enhanced frameworks such as SmartGuard [27]. These works motivate explainable pipelines that pair LLM reasoning with static/dynamic evidence—especially for composite attacks and cross-contract behaviors.

In contrast to the above, our article keeps the *attack* as the central unit and uses a program-structure-oriented taxonomy to align historic exploits, modern terminology, datasets, and detection/verification evidence. This framing simplifies mapping legacy names to unified families and clarifies which analysis capabilities are necessary to prevent each class of exploit.

3 Taxonomy of Smart Contract Vulnerabilities

As Ethereum smart contracts grow more complex and handle greater value, security risks stemming from subtle design flaws have become increasingly critical. Despite the availability of multiple vulnerability lists and static analysis tools, inconsistencies in naming, classification, and coverage persist. Some taxonomies group issues by symptoms (e.g., “reentrancy”), while others follow tool-driven patterns. However, these approaches often lack a consistent basis for identifying root causes or guiding detection and mitigation.

To address this, we propose a unified taxonomy grounded in *program structure*—how contracts process control, handle data, validate inputs, and interact with their environment. By focusing on the structural source of each vulnerability, our taxonomy helps bridge the gap between diverse historical terminology and modern detection techniques, making it easier to reason about, detect, and prevent vulnerabilities.

The Eight Root-Cause Families We classify vulnerabilities into eight families (visualized in Figure 1):

- | | |
|--|--|
| (a) Control-Flow Interaction (e.g., reentrancy) | (e) Environment Dependency (e.g., timestamp) |
| (b) External Calls & Error Handling | (f) Access Control |
| (c) State/Storage Integrity (e.g., unsafe <code>delegatecall</code>) | (g) ABI/Input Validation |
| (d) Arithmetic & Type Safety | (h) Network/Protocol Layer (e.g., replay attacks) |

Design Principles Each family is designed to be:

- **Root-cause aligned:** Groups flaws by shared structural origin.
- **Tool-compatible:** Maps directly to signals used by static, dynamic, and ML detectors.
- **Education-friendly:** Paired with minimal code, exploit trace, and fix.

Placement Criteria A vulnerability is assigned to the family that *most directly* explains its exploit and suggests its fix. We use six lightweight criteria:

- Control flow:* Does the issue involve execution order? (e.g., reentrancy)
- Storage access:* Is state aliased or corrupted? (e.g., `delegatecall`)
- Environment:* Does blockchain state affect logic? (e.g., `block.timestamp`)
- Identity:* Are permissions checked correctly? (e.g., `tx.origin`)
- Input/ABI:* Is decoding ambiguous? (e.g., packed encoding)
- Cross-domain:* Does it span chains or protocols? (e.g., signature replay)

Figure 1 organizes these criteria clockwise by reasoning complexity, from local arithmetic to cross-chain assumptions.

Impact and Utility This taxonomy enables more effective detection, auditing, and classification in both research and practice. It helps:

- Unify terminology across legacy datasets, reports, and tools;
- Clarify the intent of each detection technique by aligning it with structural causes;
- Guide developers toward better mitigation by identifying consistent patterns;

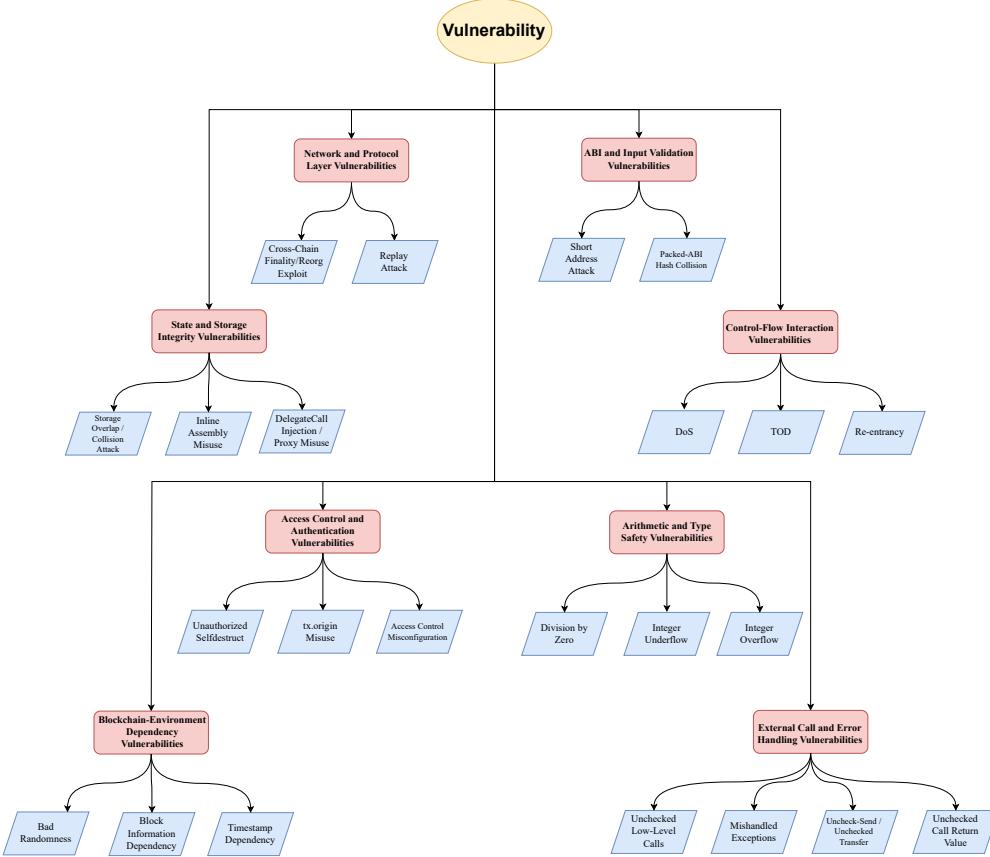


FIGURE 1. Program-structure taxonomy: eight families, common attacks, and canonical defenses.

- Provide a reliable foundation for dataset labeling, evaluation, and benchmarking.

Instead of treating vulnerabilities as isolated exceptions, this classification reveals their shared structural more robust and systematic security analysis across the entire lifecycle of smart contract development.

4 Smart Contract Vulnerabilities

Smart contracts, by design, encode and enforce rules autonomously on blockchain platforms without centralized intervention. However, this autonomy makes them especially vulnerable to logic errors, cryptoeconomic flaws, and environment-specific assumptions. In this section, we present a structured walkthrough of vulnerability classes as categorized in our taxonomy (Figure 1). Each vulnerability is analyzed in terms of its definition, technical explanation, real-world exploitation scenarios, and mitigation recommendations. The goal is to provide developers, auditors, and researchers with actionable insight into the mechanics behind recurring smart contract flaws. To enhance clarity, we group vulnerabilities into eight root-cause

families, such as control-flow mismanagement, access control misconfiguration, and reliance on unpredictable blockchain state. For each entry, we also link representative code listings and note how legacy terminology aligns with the modern classification. This consolidated view enables consistent understanding and sets a foundation for tool evaluation, security auditing, and future research in smart contract safety.

4.1 Control-Flow Interaction Vulnerabilities

This family covers flaws that break the intended execution order within or across contracts. They typically arise when developers assume atomic behavior in a non-atomic environment, allowing adversaries to interleave or reorder transactions. As shown in Figure 1, these vulnerabilities form the first layer of the taxonomy, where execution flow integrity governs all subsequent state and logic correctness.

4.1.1 Re-entrancy

Definition: Reentrancy is a vulnerability that arises when an external contract invokes a function in the victim contract before the previous execution completes. If the internal state of the contract is not updated before the external call, the attacker can exploit the recursive flow to repeatedly trigger sensitive functions such as withdrawals [11, 15].

```
contract Reentrancy {
    mapping(address => uint256) public balances;
    bool locked = false;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() public {
        require(!locked, "Reentrant call"); // Vulnerability: Reentrancy guard check exists
        locked = true;

        (bool success, ) = msg.sender.call{value: balances[msg.sender]}("");
        require(success, "Transfer failed");

        balances[msg.sender] = 0;
        // Fake unlock - conditional and unsafe
        if (balances[msg.sender] == 0) {
            locked = false; // Vulnerability: Lock released conditionally
        }
    }
}
```

Listing 1. Reentrancy Vulnerability with Fake Guard

Explanation of Vulnerability: This contract implements a `locked` flag as a reentrancy guard; however, its logic is flawed:

- **Line 6 –** Ether is transferred to `msg.sender` using a low-level `call` before the balance is cleared.
- **Line 10 and Line 19 –** Although a reentrancy guard is introduced with the `locked` flag, its reset depends on the condition `balances[msg.sender] == 0`, which is attacker-controlled. During reentrant execution, this condition may never be satisfied, allowing multiple recursive invocations.

Why This Is a Vulnerability:

- **Violation of CEI discipline:** The contract performs an external call before updating its internal state, breaking the Checks–Effects–Interactions (CEI) pattern and exposing inconsistent state during execution.

- **Synchronous execution model:** Because Ethereum executes external calls synchronously, the callee can re-enter the caller's function before the first execution completes, allowing repeated access to mutable state variables.
- **Deferred state updates:** Sensitive values such as balances are updated only after the external call, allowing attackers to reuse stale data from previous invocations.
- **Weak or conditional reentrancy guard:** Guards that depend on attacker-controlled conditions or are reset inconsistently (e.g., conditional unlocks) fail to prevent recursive entry.
- **Post-call invariant enforcement:** Logic that enforces key invariants (e.g., resetting balances) after the external interaction is vulnerable to manipulation during reentrant calls.

As a result, internal accounting and control flow can become inconsistent, enabling unauthorized withdrawals, recursive fund extraction, or logical bypasses during reentrant invocations.

Exploitation Scenario:

- An attacker deploys a contract with a fallback function that calls `withdraw()` when Ether is received.
- Upon the first `withdraw()` call, Line 10 triggers the fallback function before Line 12 clears the balance.
- The attacker recursively invokes `withdraw()` multiple times and receives Ether in each iteration.
- Since the `locked = false` statement is placed conditionally and may never execute, the reentrancy guard fails entirely.

Why the Guard is Ineffective (Fake Guard):

- A valid reentrancy guard must be enforced at the beginning and released deterministically at the end of execution.
- In this example, the unlock logic (Line 14) is based on a balance check that may not hold if recursive calls interfere, rendering the lock useless.

Mitigation Recommendations:

- Use the checks-effects-interactions pattern by placing the state update (`balances[msg.sender] = 0`) before the external `call`.
- Employ well-audited libraries such as OpenZeppelin's `nonReentrant` modifier.
- Ensure that any guard variables such as `locked` are always reset unconditionally at the end of execution.

4.1.2 Transaction Order Dependency (TOD) / Frontrunning

Definition: Transaction Order Dependence (TOD) refers to a vulnerability where the correctness or fairness of contract logic depends on the ordering of transactions in a block. Since miners choose which transactions to include and in what order, adversaries can exploit this by front-running other users' transactions [12, 21].

```
contract TxOrderDependence {
    mapping(address => uint256) public balances;

    function donate() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() public {
        // Vulnerability: Transaction ordering dependence in withdrawal
        require(balances[msg.sender] > 0);
        msg.sender.transfer(balances[msg.sender]);
        balances[msg.sender] = 0;
    }
}
```

```

    }

    function commitReveal(bytes32 hash) public {
        // Vulnerability: Unvalidated commit input, lacks sender binding
        storedHash = hash;
    }

    function reveal(uint secret) public {
        // Vulnerability: No link to original sender in commit-reveal scheme
        require(keccak256(abi.encodePacked(secret)) == storedHash);
        reward[msg.sender] = secret;
    }

    uint public nonce;
    function increment(uint input) public {
        // Warning: nonce pattern can be abused without stronger validation
        require(input == nonce);
        nonce++;
    }
}

```

Listing 2. *Transaction Ordering Dependence Vulnerability*

Explanation of Vulnerability:

- **Line 8 – Withdrawal Based on Mutable State** A user’s balance is checked and transferred in the same function, making it susceptible to frontrunning before the balance is reset.
- **Line 15 – Missing Binding in Commit Phase** The stored hash is global, not tied to a sender. Any caller can overwrite it, introducing a race condition.
- **Line 20 – Reveal Phase Unbound to Committer** The reveal logic compares input to the shared hash but does not confirm the revealing party is the one who committed. This is exploitable by frontrunners.
- **Line 29 – Weak Nonce Usage** The nonce is updated based on a public check and can be predicted or manipulated if reused or not protected contextually.

Why This Is a Vulnerability:

- **Dependence on transaction sequencing:** The correctness or fairness of the contract logic depends on the order in which transactions are included in a block. Since miners or validators can freely reorder transactions, execution outcomes become nondeterministic.
- **Mempool visibility and frontrunning:** Pending transactions are publicly visible in the mempool. Adversaries can observe these and submit similar transactions with higher gas fees to be processed first, gaining unfair advantages.
- **Shared mutable state:** Functions such as `withdraw()` rely on global state variables (`balances`) that multiple users can modify concurrently. This shared state enables race conditions when competing transactions are processed.
- **Lack of sender binding in commit–reveal patterns:** Commit–reveal schemes that do not associate the commitment with a specific sender (e.g., `storedHash`) allow attackers to overwrite or reveal values on behalf of others.
- **Predictable or weak nonce usage:** Publicly accessible counters or nonce variables can be anticipated or reused by attackers, allowing them to preempt legitimate operations or bypass intended sequencing constraints.

Such dependence on nondeterministic transaction ordering enables adversaries to front-run withdrawals, overwrite commits, or exploit race conditions—compromising fairness, consistency, and contract integrity.

Exploitation Scenario:

- A user submits a `withdraw()` call. An attacker sees it in the mempool and submits a higher-fee withdrawal that empties the balance first.
- During the `commitReveal()` phase, an attacker overwrites the committed hash. In the next block, they reveal their precomputed secret, stealing the reward.
- The `increment()` function can be replayed or bypassed if the nonce logic is used across contexts.

Mitigation Recommendations:

- Use pull-based withdrawal patterns with lock periods or commit delays to defend against frontrunning.
- Always bind commit phases to sender addresses: `commit[msg.sender] = hash`.
- Add timeouts, nonce expiry, or salt-based nonces to secure the commit-reveal pattern.
- Strengthen nonce control with role checks or context-aware validation.

4.1.3 Denial of Service (DoS)

Definition: Denial-of-Service (DoS) with revert occurs when smart contracts expose logic that can be intentionally halted or forced to fail, such as looping over external input or relying on untrusted receivers [8].

```
mapping(address => uint) public balances;
address[] public users;

function processPayments(uint iterations) public {
    for (uint i = 0; i < iterations; i++) {
        // Vulnerability: loop controlled by external argument (DoS risk)

        if (balances[users[i]] == 0) {
            revert("Zero balance");
            // Vulnerability: revert inside loop can halt all processing (DoS)
        }

        (bool success, ) = users[i].call{value: balances[users[i]]}("");
        // Vulnerability: untrusted external call in loop (DoS and reentrancy risk)
        // Vulnerability: low-level call with result unchecked before state update

        balances[users[i]] = 0;
    }
}
```

Listing 3. Denial-of-Service (DoS) Vulnerability

Explanation of Vulnerabilities:

- **Line 5 – Argument-Controlled Loop** The loop count is fully controlled by the caller through the `iterations` parameter. This allows gas griefing and unaudited unbounded loops.
- **Line 9 – Revert Inside Loop** A revert due to a user having zero balance will break the entire function, stopping all further payments.
- **Line 13 – Untrusted External Call in Loop** Calling untrusted addresses inside a loop exposes the function to reentrancy, fallback abuse, or gas griefing by external contracts.
- **Line 13 – Unchecked Result of call** The return value of `call` is not verified. Failed Ether transfers will go unnoticed, and funds may remain locked or lost.

Why This Is a Vulnerability:

- Allows a single malicious or misconfigured user to halt or block contract execution.
- Violates principles of fault isolation and resilience.
- Enables gas exhaustion attacks or silent failures that break contract assumptions.

Exploitation Scenario:

- A malicious user sets `iterations` to a high number, causing gas exhaustion.
- A zero-balance user causes revert in the loop, blocking payments to others.
- A contract with a gas-heavy fallback function is inserted into `users`, making the call fail or waste gas.

Mitigation Recommendations:

- Avoid processing logic in unbounded loops controlled by external input.
- Validate user input and avoid critical reverts inside loops.
- Use the pull-payment pattern: let users claim funds individually.
- Check all low-level call results and handle failures gracefully:

```
(bool success, ) = users[i].call{value: amount}("");
require(success, "Transfer failed");
```

- Split long operations into batches using state tracking across transactions.

4.2 External Call and Error Handling Vulnerabilities

This family captures faults where contracts mishandle results of external interactions. Low-level primitives such as `call`, `send`, and `delegatecall` return a boolean success flag instead of reverting automatically. Failing to inspect these outcomes or to isolate fallback behavior leads to silent execution failures, inconsistent state, or exploitable control flow.

4.2.1 Unchecked Call Return Value

Definition: An *unchecked call return value* vulnerability occurs when a smart contract performs a low-level external call (such as `call`, `delegatecall`, `staticcall`, or legacy `send`) but fails to verify the returned boolean result. Because these primitives do not automatically revert on failure, ignoring the return value can lead to *silent execution failure*, leaving the contract in an inconsistent state or enabling attackers to manipulate business logic [1, 2, 11, 12, 37].

```
pragma solidity ^0.8.0;

contract PaymentProcessor {
    mapping(address => uint256) public balances;
    event PaymentSent(address indexed from, address indexed to, uint256 amount);

    function deposit() external payable {
        balances[msg.sender] += msg.value;
    }

    function pay(address payable recipient, uint256 amount) external {
        require(balances[msg.sender] >= amount, "Insufficient balance");

        // Effect: balance deducted before external call
        balances[msg.sender] -= amount;

        // Vulnerability: ignoring return value of low-level call
        recipient.call{value: amount}("");

        // Continues as if transfer succeeded even if it failed
        emit PaymentSent(msg.sender, recipient, amount);
    }
}
```

Listing 4. *Unchecked call return value Vulnerability*

Explanation of the Vulnerability: In Listing 4, the vulnerable behavior resides primarily on Lines 17–18 and 15–21, where the contract performs a low-level call without verifying its outcome and continues execution under the false assumption of success.

- **Line 15 – Premature State Update** The sender’s balance is deducted *before* performing the external call. This violates the Checks–Effects–Interactions (CEI) pattern by changing state before verifying external interaction success.
- **Line 18 – Ignored Return Value** The expression `recipient.call{value: amount}("")` executes an external call but disregards the returned boolean (`success`). If the callee reverts or runs out of gas, the call silently fails.
- **Line 21 – Misleading Event Emission** Despite potential failure of the Ether transfer, Line 21 emits `PaymentSent`, signaling success even though no funds left the contract. This introduces logical inconsistency between on-chain logs and the actual state.

If the recipient’s fallback function reverts or consumes excessive gas, the transaction fails internally but the contract does not revert. The balance remains reduced while the event log reports a successful payment, resulting in accounting mismatches or permanent fund loss. This logical inconsistency can result in fund loss, denial of service, or accounting errors. Static analysis frameworks (e.g., SmartBugs, SolidiFI) often flag such patterns as part of the *External Call and Error Handling* category [11,37].

Why This Is a Vulnerability:

- **Silent failure:** If the external call fails but its return value is unchecked, the contract continues execution under the false assumption of success.
- **Broken invariants:** Internal state (such as user balances) no longer matches the actual Ether distribution, creating accounting inconsistencies.
- **Exploitable failures:** Attackers may deliberately cause reverts to exploit refund mechanisms or repeatedly trigger inconsistent states [2,12].

Exploitation Scenario:

- **Malicious recipient contract:** An attacker deploys a recipient contract whose fallback function always reverts or consumes excessive gas, ensuring that any Ether transfer to it will fail silently.
- **Ignored call result:** When the vulnerable `PaymentProcessor` contract executes `recipient.call{value: amount}("")` (Line 18), it fails to check the returned boolean value. The call reverts internally, but the contract proceeds as if successful.
- **State desynchronization:** Because the sender’s balance is already deducted (Line 15) and no revert occurs, the contract’s internal accounting no longer reflects its actual Ether balance.
- **False success signal:** The `PaymentSent` event (Line 21) is still emitted, misleading observers and off-chain systems into believing that payment succeeded.
- **Exploit chain reaction:** The attacker can exploit this inconsistency by repeatedly triggering refunds, forcing compensatory logic, or leveraging mismatched balances to drain funds or lock assets.

Mitigation Recommendations:

- **Always verify the return value:**

```
(bool success, ) = recipient.call{value: amount}("");
require(success, "Transfer failed");
```

- **Adopt the pull-payment pattern:** Instead of pushing funds via external calls, record user balances and let recipients withdraw them manually through a controlled `withdraw()` function [12].
- **Use the Checks–Effects–Interactions (CEI) pattern:** Perform all validations and internal state updates before making any external calls, and guard against reentrancy.

- **Use verified libraries:** Rely on well-tested frameworks like OpenZeppelin’s `PullPayment` or `PaymentSplitter` for handling Ether transfers safely.
- **Handle potential call failures gracefully:** Include fallback logic, retry mechanisms, or event logs that record unsuccessful payments for manual review [11,37].

Notes: Unchecked call return values are among the most frequent smart contract weaknesses detected by both static and dynamic analysis tools. Despite their apparent simplicity, they often indicate improper error-handling discipline and can be precursors to critical financial inconsistencies, especially in payment systems and token contracts.

4.2.2 *Uncheck-Send / Unchecked Transfer*

Definition: An *unchecked send* (or unchecked transfer) occurs when a contract transfers Ether to an external address using low-level primitives such as `call`, `send`, or `transfer` without properly checking the outcome or without constraining when and to whom transfers occur. This includes (a) ignoring the boolean success return from low-level calls, and (b) performing unconditional transfers (e.g., always sending a fixed amount to `msg.sender`) that may be abused by an attacker or cause unexpected state inconsistencies [2,11,12,37].

```
pragma solidity >=0.4.21 <0.6.0;

contract DocumentSigner {
    function bug_unchk_send7() public payable {
        msg.sender.transfer(1 ether);
        //Vulnerability: unconditional transfer without authorization or state update
    }

    mapping(bytes32 => string) public docs;

    function bug_unchk_send23() public payable {
        msg.sender.transfer(1 ether);
        // [Line 10] Vulnerability: no access control or success check
    }

    mapping(bytes32 => address[]) public signers;

    function bug_unchk_send31() public payable {
        msg.sender.transfer(1 ether);
        //Vulnerability: repetitive value transfer without validation or accounting
    }

    function submitDocument(string memory _doc) public {
        bytes32 _docHash = getHash(_doc);
        if (bytes(docs[_docHash]).length == 0) {
            docs[_docHash] = _doc;
            emit NewDocument(_docHash);
        }
        //Safe logic; unrelated to transfer vulnerability
    }

    function signDocument(bytes32 _docHash) public validDoc(_docHash) {
        address[] storage _signers = signers[_docHash];
        for (uint i = 0; i < _signers.length; i++) {
            if (_signers[i] == msg.sender) return;
        }
        _signers.push(msg.sender);
        //benign functionality; included for context
    }
}
```

Listing 5. Unchecked transfers Vulnerability

Explanation of the vulnerability: In Listing 5, several functions transfer Ether directly to the caller without verification or access control.

Line 4 - The function `bug_unchk_send7()` performs an unconditional `msg.sender.transfer(1 ether)` call without checking authorization or success, allowing any user to withdraw Ether arbitrarily.

Line 11 - The function `bug_unchk_send23()` repeats the same pattern, again omitting success

handling and enabling silent failures if the transfer reverts due to gas or fallback execution.

Line 18 - The function `bug_unchk_send31()` further illustrates unsafe code reuse. Each of these functions can be exploited independently to drain funds since no accounting or role validation is enforced.

Why This Is a Vulnerability:

- **Immediate fund leakage:** Any caller can repeatedly invoke these functions to drain Ether since transfers occur without accounting updates or restrictions.
- **Silent failure and broken invariants:** If a transfer fails (e.g., due to insufficient gas or receiver revert), the contract continues execution as though successful, leading to discrepancies between recorded and actual balances.
- **Gas and griefing exposure:** Older `transfer/send` semantics forward only 2300 gas. Attackers can exploit gas-heavy fallback functions to trigger failures or unexpected state behavior [2, 11].

Exploitation scenario:

- An attacker repeatedly calls any of the `bug_unchk_send*` functions to withdraw one ether per call until the contract balance reaches zero. No authentication or balance accounting prevents this drain.
- Alternatively, if a benign recipient has a gas-intensive fallback function, the transfer fails silently, leaving state inconsistent while the contract assumes success—opening room for double spends or accounting manipulation.

Mitigation recommendations:

- **Avoid unconditional transfers:** Always link transfers to verified conditions (`onlyOwner`, `balances[msg.sender] > amount`) and perform updates before external calls.
- **Prefer pull-over-push payments:** Let users withdraw via a controlled `withdraw()` function rather than automatic transfers, minimizing reentrancy and gas griefing.
- **Check return values explicitly:**

```
(bool success, ) = payable(recipient).call{value: amount}("");
require(success, "Transfer failed");
```

- **Use secure libraries:** Employ verified payment abstractions such as OpenZeppelin's `PullPayment` or `PaymentSplitter`.
- **Enforce access control:** Add `onlyOwner` or role-based modifiers and rate limits to all Ether transfer functions.
- **Document expected failure behavior:** Handle benign failures gracefully (refunds, retries) and audit all unchecked send warnings flagged by Static Application Security Testing (SAST) tools.

Notes:

- `transfer()` and `send()` historically forwarded a fixed 2300 gas stipend; in modern environments using `callvalue:...("")` combined with an explicit ‘`require(success)`’ is the recommended approach, but it must be paired with CEI and reentrancy protections [2].
- Static analysis datasets (SmartBugs, SolidiFI) and recent detector evaluations frequently mark unchecked sends as a distinct family of defects because they are both common and high-impact; prioritize fixing them in audits and test harnesses [11, 12].

4.2.3 Mishandled Exceptions

Definition: Mishandled exceptions occur when errors such as failed external calls, invalid inputs, or logic exceptions are not properly checked or recovered from. This leads to silent failures or wasted gas and hinders reliable contract behavior [14,38].

```
contract MishandledExceptions {
    function unsafeDivision(uint a, uint b) public pure returns (uint) {
        return a / b; // Vulnerability: division without zero check
    }

    function callWithoutRevert(address target) public {
        (bool success, ) = target.call("");
        if (!success) {
            // Vulnerability: external call failure ignored
        }
    }

    function withEmptyCatch() public {
        try this.unsafeDivision(10, 0) {
        } catch {
            // Vulnerability: empty catch block without handling logic
        }
    }

    function useAssert(uint a) public pure {
        assert(a != 0); // Vulnerability: assert used instead of require for input validation
    }
}
```

Listing 6. *Mishandled Exceptions Vulnerability*

Explanation of Vulnerability:

- **Line 3 – Unsafe Division Without Check**

There is no check that `b` is non-zero before performing division, risking a crash due to divide-by-zero.

- **Line 9 – Unhandled Call Failure**

The call result is checked but ignored. There is no revert, log, or handling, leading to hidden failures.

- **Line 15 – Empty Catch Block**

The catch block is syntactically present but lacks logic. This suppresses errors that could indicate major issues.

- **Line 21 – Misuse of assert**

The `assert` statement is used for input validation, which is incorrect. Assertions are for developer invariants and consume all remaining gas on failure.

Why This Is a Vulnerability:

- Exceptions are part of critical control flow. Ignoring or misusing them makes contract behavior unreliable.
- Developers and auditors may not detect silent failures.
- Misuse of `assert()` can crash the contract with high gas loss and no user-facing error.

Exploitation Scenario:

- An attacker could exploit silent failure to bypass logic intended to restrict behavior.
- Calls that appear to succeed may be quietly failing, leading to loss of funds or broken logic.
- Misused `assert()` can be weaponized to intentionally waste gas and disable contract functionality.

Mitigation Recommendations:

- Always perform zero checks before division.
- Revert explicitly after failed external calls:

```
(bool success, ) = target.call("");
require(success, "Call failed");
```

- Avoid empty `catch` blocks. Add logs or safe fallbacks.
- Replace `assert()` with `require()` for all input validations.

4.2.4 Unchecked Low-Level Calls

Definition: Unchecked send occurs when a contract sends Ether or executes low-level calls using `call()`, `delegatecall()`, or `send()` without verifying whether the call succeeded. This may result in silent failure and unintended behavior, including loss of funds or unsafe execution paths [37].

```
contract UncheckedCalls {
    function withdraw(address payable user, uint amount) public {
        user.call{value: amount}("");
    }

    function sendEther(address payable recipient, uint amount) public {
        (bool success, ) = recipient.call{value: amount}("");
        // success is ignored, not checked
    }

    function unsafeDelegate(address target, bytes memory data) public {
        (bool ok, ) = target.delegatecall(data);
        if (ok) {
            // do something
        }
        // missing revert on failure
    }
}
```

Listing 7. Unchecked Low-Level Calls Vulnerability

Explanation of Vulnerability:

- **Line 3 – Unchecked call** - Ether is sent using `user.call` without capturing or checking the result, potentially leading to untracked failures.
- **Line 7 – Call With Ignored Success Flag** - Although the success flag is captured, it is never validated. This is functionally equivalent to ignoring the outcome.
- **Lines 11–13 – Silent delegatecall Failure** - The contract checks the result of the delegate call but fails to revert on failure. This allows execution to continue even if a critical subroutine fails.

Why This Is a Vulnerability:

- Solidity's low-level calls return a success boolean instead of throwing an exception.
- Ignoring this boolean leads to logic that assumes success even when the call failed.
- This can break invariants, silently lose Ether, or create exploitable conditions in contract control flow.

Exploitation Scenario:

- An attacker could manipulate gas costs or fallback logic to make the call fail.
- The contract would continue under the false assumption that the Ether was transferred or the delegate executed safely.
- This allows denial of service or bypass of logical checks.

Mitigation Recommendations:

- Always verify the return value of low-level calls:

```
(bool success, ) = user.call{value: amount}("");
require(success, "Call failed");
```

- Replace low-level calls with safer abstractions when possible, such as `transfer()`.

- Use explicit `revert()` or error handling when failures occur to maintain predictable behavior.

4.3 State and Storage Integrity Vulnerabilities

This family concerns violations of storage consistency—when two pieces of code inadvertently share or corrupt the same storage context. Typical causes include unsafe use of `delegatecall`, inconsistent layout between proxy and implementation, or unchecked inline assembly writes. As indicated in Figure 1, storage-integrity faults form the boundary between logical correctness and authorization, often leading to full contract takeover.

4.3.1 DelegateCall Injection

Definition: DelegateCall Injection arises when a smart contract uses `delegatecall` with an untrusted or user-controlled target. Since `delegatecall` runs the callee’s code in the storage context of the calling contract, any injected logic can manipulate critical state variables or hijack control [8, 16].

```
contract Proxy {
    address public implementation;

    function setImplementation(address _impl) public {
        implementation = _impl; // Vulnerability: unprotected setter for delegatecall target
    }

    fallback() external payable {
        // Vulnerability: delegatecall without validation or access control
        (bool success, ) = implementation.delegatecall(msg.data);
        // Vulnerability: raw msg.data forwarded without filtering
        // Vulnerability: no whitelist or function selector check
        require(success, "delegatecall failed");
    }
}
```

Listing 8. *DelegateCall Injection Vulnerability*

Explanation of Vulnerability:

- **Line 5 – Unprotected Target Setter** The `setImplementation()` function can be invoked by anyone, allowing replacement of the `implementation` contract with malicious logic.
- **Line 8 – Unsafe Use of delegatecall** The fallback function blindly uses `delegatecall` on user-controlled `msg.data` without function signature filtering, authentication, or validation.
- **Global – No Whitelist or Caller Control** The proxy contract has no access control, condition checks, or contract validation mechanisms.

Why This Is a Vulnerability:

- **Execution in caller’s storage context:** The `delegatecall` instruction executes the callee’s code in the storage context of the caller. This means any state changes made by the callee directly affect the proxy’s storage, including critical variables such as ownership or balances.
- **Unrestricted target modification:** Without access control on `setImplementation()`, any user can redirect the proxy to a malicious implementation, giving them full control over the proxy’s logic and state.
- **Arbitrary code execution:** A malicious implementation can perform unauthorized actions such as overwriting privileged storage slots, draining Ether, or invoking `selfdestruct()`, effectively taking over or destroying the contract [8, 16].

- **Lack of input validation and function filtering:** The fallback function forwards raw `msg.data` without validating function selectors or ensuring that the target code is trusted. This enables attackers to craft arbitrary payloads for storage corruption or privilege escalation.
- **Immediate and irreversible impact:** Because all changes occur within the proxy's storage, exploitation results in direct and permanent corruption of the contract's state—making recovery impossible once the malicious code executes.

Exploitation Scenario:

- An attacker deploys a malicious contract with logic to overwrite storage, drain funds, or selfdestruct.
- They invoke `setImplementation()` to point the proxy at the malicious contract.
- Using crafted `msg.data`, the attacker triggers arbitrary execution inside the proxy's context.

Mitigation Recommendations:

- Restrict `setImplementation()` using access control (e.g., `onlyOwner`).
- Validate new implementation contracts (e.g., check for minimum code size or trusted bytecode).
- Avoid forwarding raw `msg.data`; use explicit routing with known selectors.
- Follow hardened patterns like OpenZeppelin's `TransparentUpgradeableProxy`.

4.3.2 Inline Assembly Misuse

Definition: *Inline assembly misuse* refers to security faults introduced by using Yul/assembly to read or write memory/storage or to perform low-level calls without the type, bounds, and exception safety that Solidity normally enforces. Typical issues include unchecked `mload/calldataload`, arbitrary `sstore` to privileged slots, clobbering of storage layout, and ignoring call return values [2, 4, 7].

```
pragma solidity ^0.8.20;

contract AssemblyPitfalls {
    // Critical state
    address public owner; // slot 0
    uint256 public balance; // slot 1

    constructor() { owner = msg.sender; }

    // 1) Unchecked read from calldata into arbitrary storage slot
    function setArbitrary(bytes calldata raw) external {
        assembly {
            // Vulnerability: no length check; reads past end if raw < 32 bytes
            let slot := calldataload(raw.offset)
            let val := calldataload(add(raw.offset, 32))
            // Vulnerability: attacker-controlled sstore -> clobbers any slot (e.g., slot 0 owner)
            sstore(slot, val)
        }
    }

    // 2) Low-level call that ignores success and returndata
    function unsafeCall(address target, bytes calldata data) external {
        assembly {
            let ok := call(gas(), target, 0, data.offset, data.length, 0, 0)
            // Vulnerability: 'ok' ignored; execution continues even if target reverted
        }
    }

    // 3) Miscomputed slot via keccak; off-by-one corrupts adjacent mapping
    mapping(bytes32 => uint256) public limits; // layout uses keccak(key . slot)
    function setLimit(bytes32 key, uint256 v) external {
        assembly {
            // slot of 'limits' is 2, but we mistakenly use 3 -> writes foreign state
            mstore(0x00, key)
            mstore(0x20, 3) // Vulnerability: wrong parent slot (should be 2)
        }
    }
}
```

```

        let slot := keccak256(0x00, 0x40)
        sstore(slot, v)
    }
}

```

Listing 9. *Unsafe inline assembly Vulnerability*

Explanation of the vulnerability: Listing 9 presents three misuse cases of inline assembly that compromise storage integrity and reliability.

- **Lines 11–17** - The function `setArbitrary()` performs unchecked `calldataload` operations on attacker-controlled input and directly executes `sstore(slot, val)`. This allows arbitrary writes to any storage slot, including critical variables such as `owner`.
- **Lines 22–26** - The function `unsafeCall()` executes a low-level `call` but ignores its success flag, enabling silent failures and masking reverted external executions.
- **Lines 31–39** - The function `setLimit()` miscomputes a mapping slot by writing to the wrong parent index (3 instead of 2), causing state overlap and corruption of unrelated variables.

Together, these errors show how small oversights in assembly—unchecked memory reads, ignored return values, and misaligned storage math—can escalate into full contract compromise.

Why This is a Vulnerability:

- Assembly bypasses Solidity’s type checks, bounds checks, and storage layout guarantees; small mistakes yield arbitrary write primitives.
- Ignoring return values from low-level calls reintroduces failure-oblivious control flow.
- Miscomputed slots or offsets corrupt unrelated variables or mappings, breaking invariants and enabling privilege escalation [4,7].

Exploitation Scenario:

- An attacker calls `setArbitrary` with `slot=0` and `val=attacker`, resetting `owner`. They then take over administrative functions or drain funds.
- Separately, `unsafeCall` can be used to mask downstream failures and leave accounting in an inconsistent state.

Mitigation Recommendations:

- Avoid inline assembly unless necessary; prefer safe Solidity primitives and libraries.
- If assembly is required, validate lengths/offsets before any `mload/calldataload`; encapsulate slot math in audited helpers and add invariants/comments.
- Always check and propagate the return value of `call/delegatecall`; bubble up revert reasons where possible.
- Lock down critical slots (e.g., use ERC-1967 fixed slots for proxy admin/implementation) and add runtime asserts that guard ownership variables.
- Use static analyzers and bytecode-level CFG/storage tools to catch slot math mistakes and unchecked calls [8,11].

4.3.3 Storage Overlap / Collision Attack

Definition: A *storage overlap/collision* occurs when contracts that share a storage context use inconsistent layouts, causing one contract to read/write slots assumed by another. In upgradeable patterns this can overwrite admin/owner pointers or corrupt application state [4,7].

```

pragma solidity ^0.8.20;

// Minimal proxy: stores admin at slot 0, implementation at slot 1
contract MinimalProxy {
    address public admin; // slot 0
    address public implementation; // slot 1
}

```

```

constructor(address impl) { admin = msg.sender; implementation = impl; }

fallback() external payable {
    address impl = implementation;
    assembly {
        calldatcopy(0, 0, calldatasize())
        let ok := delegatecall(gas(), impl, 0, calldatasize(), 0, 0)
        returndatcopy(0, 0, returndatasize())
        switch ok
        case 0 { revert(0, returndatasize()) }
        default { return(0, returndatasize()) }
    }
}

// First implementation uses slot 0 for 'owner' -> collides with proxy.admin
contract ImplV1 {
    // !!! Collision: this variable sits at slot 0 in the shared storage
    address public owner; // slot 0 (overlaps MinimalProxy.admin)

    function init() external {
        owner = msg.sender; // Vulnerability: overwrites proxy.admin (slot collision)
    }

    function doAdminThing() external {
        require(msg.sender == owner, "not owner");
        // privileged action...
    }
}

// Safer layout: reserve slots or use ERC-1967 fixed slots + storage gaps
contract ImplV2 {
    // storage gap to avoid colliding with proxy slots (example pattern)
    uint256[50] private __gap;
    address public owner; // now at a higher, non-conflicting slot

    function init(address o) external { owner = o; }
}

```

Listing 10. Proxy storage collision Vulnerability

Explanation of Vulnerability: In Listing 10, the vulnerability arises due to overlapping storage slots between the proxy and its implementation contract:

- **Line 5 - Proxy Admin Declaration** - The MinimalProxy contract stores the `admin` variable at slot 0.
- **Line 26 - Implementation Variable Collision** - The `ImplV1` contract declares `owner` as its first state variable, which also occupies slot 0, colliding with `proxy.admin`.
- **Line 29 - Overwriting Critical Storage** - In `ImplV1.init()`, the statement `owner = msg.sender;` (see inline vulnerability comment) writes directly to slot 0 in the proxy's storage, overwriting the admin's address with the attacker's.
- Because `delegatecall` on **Line 14** executes `ImplV1` in the proxy's storage context, the overwrite occurs transparently, granting the attacker full administrative privileges.

Why This is a Vulnerability:

- Shared storage with inconsistent layouts creates unintended aliasing between privileged pointers and application variables.
- Upgrades that reorder, insert, or change types of state variables shift slot indices, breaking invariants across versions.
- The effect is silent at compile time; the fault manifests only at runtime through `delegatecall`, often after deployment [7, 31].

Exploitation Scenario: An attacker triggers `ImplV1.init()` via the proxy (or any function that writes to slot 0), setting `owner=attacker` and thereby overwriting `proxy.admin`. With admin control, they upgrade to a malicious implementation and drain funds or brick the system.

Mitigation Recommendations:

- Adopt ERC-1967/Transparent/UUPS patterns that store admin/implementation in fixed, hashed slots (e.g., `bytes32(uint256(keccak256("eip1967.proxy.admin"))-1))`).
- Preserve layout across upgrades: never reorder/remove variables; only append; use *storage gaps* (`uint256[N] _gap`) to future-proof layouts [4].
- Add upgrade guards: access-controlled `onlyProxy/onlyAdmin`, code-hash/size checks on new implementations, and initialization modifiers that can be called only once.
- Employ static analysis and differential bytecode tools to compare layouts pre/post-upgrade and detect potential slot collisions [8, 11].
- Consider formal specs for critical slots (admin/implementation/pauser) and assert them in on-chain getters or off-chain monitors to catch drift early [31].

4.4 Arithmetic and Type Safety Vulnerabilities

This family groups numerical and type-safety faults—overflows, underflows, and division-by-zero—that break arithmetic invariants. Before Solidity 0.8.0, arithmetic operations wrapped silently, enabling balance or time-lock bypasses. These issues underpin financial logic correctness and remain detectable patterns in legacy codebases.

4.4.1 Integer Overflow

Definition: An *integer overflow* vulnerability occurs when an arithmetic operation on an unsigned or signed integer exceeds its maximum storable value. In Solidity versions prior to 0.8.0, this causes the value to wrap around to zero (modular arithmetic), potentially breaking contract logic or allowing unauthorized fund access [1, 2, 11, 12, 37].

```
pragma solidity >=0.4.21 <0.6.0;

contract DocumentSigner {
    mapping(address => uint) public lockTime_intou13;

    function increaseLockTime_intou13(uint _secondsToIncrease) public {
        // Vulnerability: addition without overflow check
        lockTime_intou13[msg.sender] += _secondsToIncrease; // overflow possible
    }

    function withdraw_intou13() public {
        require(now > lockTime_intou13[msg.sender]);
        uint transferValue_intou13 = 10;
        msg.sender.transfer(transferValue_intou13);
    }

    function bug_intou36(uint8 p_intou36) public {
        uint8 vundflw1 = 0;
        vundflw1 = vundflw1 + p_intou36; // overflow example
    }
}
```

Listing 11. *integer overflow Vulnerability*

Explanation of Vulnerability: In Listing 11 the root faults are concrete, local arithmetic writes that can be driven by attacker-controlled inputs and that silently wrap in pre-0.8.0 Solidity. We point to the specific vulnerable lines and show inline vulnerability comments in the code (see listing) so readers and static tools can quickly locate the problem:

- **Line 8 - Unchecked addition into storage (overflow primitive)** `lockTime_intou13[msg.sender] += _secondsToIncrease;` is performed without any bounds check. If the stored value is near `type(uint).max` a sufficiently large `_secondsToIncrease` will wrap the value modulo 2^{256} , producing a small timestamp and effectively resetting the lock. (See the inline comment in the listing: `// Vulnerability: addition without overflow check.`)

- **Line 12 - Time comparison uses possibly-wrapped value** `require(now > lockTime_intou13[msg.sender]);` compares the current time against a storage slot that can be attacker-manipulated by the overflow at Line 8. After overflow the predicate can be made true immediately, bypassing the intended time-lock. (Also note: `now` is an alias for `block.timestamp` in older Solidity; the logical point is that a wrapped timestamp breaks the time check.)
- **Line 14 - Transfer after weakened check** `msg.sender.transfer(transferValue_intou13);` sends funds conditioned on the (now bypassed) time check. Because the prior arithmetic can force the require to pass, this transfer executes prematurely.
- **Line 19 - Small-width integer overflow example (local/stack)** `vundflw1 = vundflw1 + p_intou36;` demonstrates the same class of bug for an 8-bit variable: additions that exceed `uint8` wrap to small values, which in real contracts can translate into corrupted counters, allowance arithmetic, or token balances. (See inline comment: `// overflow example.`)

Why This is a Vulnerability:

- Overflow enables breaking of logical invariants (e.g., time locks or balances) by manipulating arithmetic results.
- Because Solidity versions prior to 0.8.0 perform unchecked arithmetic by default, overflowed values silently wrap around instead of reverting.
- Attackers can exploit this to bypass authorization checks, manipulate storage variables, or unlock restricted resources [2, 11].

Exploitation Scenario: An attacker calls `increaseLockTime_intou13()` with a large value such that `lockTime_intou13[msg.sender]` overflows to a small number. The condition `now > lockTime_intou13[msg.sender]` in `withdraw_intou13()` then immediately evaluates to true, allowing premature withdrawal of locked Ether. This type of exploit has historically affected multiple time-lock and token contracts before the release of Solidity 0.8.0.

Mitigation Recommendations:

- **Upgrade Solidity compiler:** Use Solidity $\geq 0.8.0$, which automatically reverts on overflow and underflow.
- **Use SafeMath library:** For older compilers, import `SafeMath` from OpenZeppelin to perform checked arithmetic:

```
using SafeMath for uint256;
lockTime[msg.sender] = lockTime[msg.sender].add(_secondsToIncrease);
```

- **Validate inputs:** Restrict `_secondsToIncrease` to reasonable bounds via `require()` statements.
- **Perform fuzz testing:** Use automated tools to test arithmetic edge cases and verify no overflow occurs [12, 37].

4.4.2 Integer Underflow

Definition: An *integer underflow* occurs when a subtraction operation results in a value below zero in unsigned integer arithmetic. Before Solidity 0.8.0, this caused the value to wrap around to the maximum value representable by the type (e.g., $2^{256} - 1$ for `uint256`), leading to unintended or exploitable outcomes [1, 2, 11].

```
pragma solidity >=0.4.21 <0.6.0;

contract DocumentSigner {
    function bug_intou39() public {
        uint8 vundflw = 0;
        vundflw = vundflw - 10; // underflow bug
    }
}
```

```

    }

    function bug_intou31() public {
        uint8 vundflw = 0;
        vundflw = vundflw - 10; // underflow bug
    }

    function bug_intou35() public {
        uint8 vundflw = 0;
        vundflw = vundflw - 10; // underflow bug
    }

    function bug_intou27() public {
        uint8 vundflw = 0;
        vundflw = vundflw - 10; // underflow bug
    }
}

```

Listing 12. integer underflow Vulnerability

Explanation of Vulnerability: In Listing 12, each function demonstrates an unchecked subtraction operation on unsigned integers, leading to an *underflow* when the left operand is smaller than the right operand. Below we reference the precise vulnerable lines and explain the fault mechanism for each instance:

- **Line 5** - The variable `vundflw` is initialized to 0 and then decremented by 10. The statement `vundflw = vundflw - 10;` (see inline comment `// underflow bug`) causes the value to wrap around from 0 to 246 ($2^8 - 10$) because unsigned integers cannot represent negative values.
- **Line 10** - The same operation is repeated: `vundflw = vundflw - 10;` subtracts from zero without any validation, again producing a large wrapped value rather than reverting.
- **Line 15** - The local variable underflows in the same way, representing a repeated unsafe arithmetic pattern typical of pre-0.8.0 Solidity codebases.
- **Line 20** - The final function mirrors the same issue; the subtraction is executed without bounds checking, creating a silent wrap-around that could, in real-world code, inflate balances or counters to unrealistic values.

In pre-0.8.0 Solidity, such arithmetic does not revert, producing a large positive value rather than throwing an error. In practical contracts, similar operations on balances or counters can cause major logical inconsistencies or unauthorized token creation [2, 11, 12].

Why This is a Vulnerability:

- Underflow can turn small values (like 0) into extremely large numbers, granting attackers unintended privileges or access to funds.
- The silent wrap-around behavior enables bypassing of conditional checks and incorrect balance or time computations.
- If exploited in payment or staking contracts, attackers may withdraw excessive funds or trigger logic that depends on low numeric thresholds.

Exploitation Scenario: Suppose a balance tracking contract uses unsigned integers and allows subtraction without bounds checking:

```
balances[msg.sender] -= withdrawalAmount;
```

If `balances[msg.sender]` is smaller than `withdrawalAmount`, the result underflows to a massive number, granting the attacker a huge balance and allowing them to drain the contract. In `DocumentSigner`, calling `bug_intou39()` or `bug_intou31()` demonstrates the same arithmetic wrap-around effect.

Mitigation Recommendations:

- **Use Solidity 0.8.0 or higher:** Modern versions automatically revert on arithmetic underflow/overflow.
- **Apply SafeMath subtraction:**

```
using SafeMath for uint256;
balance = balance.sub(amount);
```

- **Enforce preconditions:** Always verify that operands maintain expected bounds before subtraction:

```
require(balance >= amount, "Insufficient balance");
```

- **Static analysis and fuzz testing:** Integrate automated analyzers such as Mythril, Slither, or SmartBugs to detect integer underflow patterns early in the development cycle [11,37].

4.4.3 Division by Zero

Definition: A *division-by-zero* vulnerability occurs when a smart contract performs an arithmetic division or modulo operation where the divisor is zero or can become zero due to user input or state manipulation. In Solidity versions prior to 0.8.0, such an operation causes a runtime exception and reverts the transaction, potentially halting dependent logic or enabling denial-of-service conditions. Although modern compilers revert safely, unvalidated user inputs or indirect zero divisors can still lead to broken logic, gas wastage, or DoS conditions [1,2,11,12,37].

```
pragma solidity ^0.5.0;

contract RewardDistributor {
    mapping(address => uint256) public rewards;
    uint256 public totalParticipants;

    function addParticipant(address participant, uint256 reward) public {
        rewards[participant] = reward;
        totalParticipants += 1;
    }

    function averageReward() public view returns (uint256) {
        // Vulnerability: possible division by zero
        // If no participants have been added, totalParticipants == 0
        return (address(this).balance) / totalParticipants;
    }

    function distribute() public {
        uint256 avg = averageReward(); // potential revert if divisor == 0
        // Distribution logic continues here...
    }
}
```

Listing 13. Division-by-zero vulnerability

Explanation of Vulnerability: In Listing 13 the bug is a single unchecked divisor used in a division operation — this is called a division-by-zero and will revert at runtime when the divisor equals zero. Key pointers:

- **Line 15 -** `return (address(this).balance) / totalParticipants;` performs division by `totalParticipants` without guaranteeing > 0 .
- **Line 19 -** `uint256 avg = averageReward();` invoking the vulnerable line causes the call to revert and abort the transaction when `totalParticipants == 0`, allowing DoS of distribution logic.
- **Lines 7-10 -** `addParticipant(...)` increments `totalParticipants`, but there is no precondition ensuring it has been called before `averageReward()`, so the divisor can legitimately be zero.

This halts execution and reverts the entire transaction, affecting all dependent logic and potentially freezing automated distribution systems. Static analyzers categorize this flaw under *Arithmetic and Type Safety* vulnerabilities [11, 37].

Why This is a Vulnerability:

- **Runtime exception:** Division or modulo by zero throws a runtime error that reverts state changes and consumes gas unnecessarily.
- **Denial of Service (DoS):** Attackers or users can deliberately trigger zero divisors to halt execution and disrupt business logic or time-based reward systems.
- **Inconsistent invariants:** Dependent calculations relying on unchecked divisors (e.g., averages, ratios, or percentages) can produce undefined or reverted results, undermining contract reliability [2].

Exploitation Scenario: Consider a reward or staking contract where an administrative function calls `averageReward()` periodically. If an attacker invokes `distribute()` before any participants are registered (i.e., `totalParticipants == 0`), the division operation reverts and halts the transaction. Repeated exploitation can render the contract temporarily unusable, blocking reward distribution or freezing dependent automated jobs. In older smart contracts, such behavior could even cause gas exhaustion loops in dependent contracts that call the vulnerable function.

Mitigation Recommendations:

- **Validate divisor inputs:** Always check that divisors are non-zero before performing division or modulo operations:

```
require(totalParticipants > 0, "Division by zero");
uint256 average = totalRewards / totalParticipants;
```

- **Apply defensive programming:** Use conditional returns to handle zero cases gracefully instead of letting execution revert:

```
if (totalParticipants == 0) return 0;
```

- **Use Solidity 0.8.0 or higher:** Modern compilers automatically revert on division-by-zero but validation should still be explicit for logical clarity and to avoid revert-based DoS chains.
- **Perform thorough unit and fuzz testing:** Test arithmetic functions with boundary values (e.g., divisor = 0) to ensure contracts handle zero divisions gracefully [12, 37].
- **Static and dynamic analysis:** Include this check in automated pipelines (Mythril, Slither, SmartBugs) as part of the Arithmetic Safety rule set.

4.5 Blockchain-Environment Dependency Vulnerabilities

Contracts that depend directly on mutable blockchain parameters—such as timestamps, block numbers, or difficulty—inherit the uncertainty of those values. This family captures vulnerabilities where environment-controlled data influence critical decisions like randomness or deadlines, enabling subtle but systematic bias by miners or validators.

4.5.1 Timestamp Dependency

Definition: Timestamp dependence occurs when a smart contract uses `block.timestamp` in critical decision-making, such as reward distribution or game deadlines. Since miners can slightly manipulate timestamps within a permissible range, attackers may influence contract logic in their favor [36].

```

contract Lottery {
    address public winner;
    uint public closeTime;

    constructor() {
        closeTime = block.timestamp + 1 days;
    }

    function drawWinner() public {
        require(block.timestamp >= closeTime, "Too early");
        winner = msg.sender; // Vulnerability: anyone can call at the right time
    }
}

```

Listing 14. *Timestamp Dependency Vulnerability*

Explanation of Vulnerability:

- **Line 10** - The function `drawWinner()` depends on `block.timestamp` to determine whether it is time to finalize the lottery. However, miners can influence the timestamp by several seconds, enabling unfair advantage.
- **Line 11** - There is no randomness or user verification—any account that calls the function after the threshold becomes the winner.

Why This Is a Vulnerability:

- Miners can include this transaction in a block with a biased `timestamp`, satisfying the `require()` condition earlier or later than intended.
- In timing-critical applications (e.g., lotteries, auctions), this subtle control creates an unfair and manipulable environment.

Exploitation Scenario:

- A miner, or an attacker working with one, monitors the block's timestamp range and submits a transaction when the contract will permit it.
- By controlling inclusion time, the attacker ensures they are the first to invoke `drawWinner()` and claim rewards.

Mitigation Recommendations:

- Avoid using `block.timestamp` for critical control logic, especially where fairness or randomness is involved.
- For randomness, use verifiable off-chain sources like Chainlink VRF.
- For deadlines, consider tolerating timestamp manipulation or using block numbers with defined intervals.

4.5.2 Block Information Dependency

Definition: This vulnerability arises when developers attempt to generate randomness using predictable blockchain variables such as `block.timestamp`, `block.difficulty`, or `blockhash`. These values can be influenced or anticipated by miners and are therefore insecure sources for randomness in smart contracts [34, 36].

```

contract RandomnessBlockInfo {
    function useTimestamp() public view returns (uint) {
        // Vulnerability: block.timestamp can be manipulated by miners
        return uint(keccak256(abi.encodePacked(block.timestamp)));
    }

    function useMultipleBlockVars() public view returns (uint) {
        // Vulnerability: using predictable values like block.difficulty and block.coinbase
        return uint(keccak256(abi.encodePacked(block.difficulty, block.coinbase)));
    }

    function useBlockhash(uint index) public view returns (bytes32) {
        // Vulnerability: blockhash is deterministic and not suitable for randomness
    }
}

```

```

        return blockhash(block.number - index);
    }
}

```

Listing 15. *Block Information Dependency*

Explanation of Vulnerability: This contract demonstrates three distinct insecure practices:

- **Line 4 – Use of `block.timestamp`** Miners can manipulate the block timestamp slightly to influence outcomes, especially in games and lotteries.
- **Line 9 – Use of `block.difficulty` and `block.coinbase`** These values are also predictable and do not offer entropy suitable for randomness.
- **Line 14 – Use of `blockhash`** The blockhash of past blocks is publicly available and deterministic. It does not provide true unpredictability.

Why This is a Vulnerability: Blockchain state variables such as `block.timestamp`, `block.difficulty`, and `blockhash` are accessible to both the contract and the miner but are neither cryptographically random nor immutable during mining. This predictability permits strategic manipulation by miners or attackers to bias outcomes that rely on these values. In randomness-sensitive applications—like lotteries, auctions, and entropy-based minting—this breaks the fairness guarantees expected by participants and may cause economic loss [14, 36].

Exploitation Scenario:

- A smart contract uses `block.timestamp` or `blockhash` to select a reward recipient.
- A miner observes the contract logic and chooses to include or exclude certain transactions to bias the outcome.
- The miner manipulates the timestamp or delays a block to maximize their chance of winning.

Mitigation Recommendations:

- Avoid using blockchain properties for randomness in production logic.
- Use secure external sources such as Chainlink VRF for verifiable randomness.
- Document any pseudo-random logic clearly if it is non-critical or only used for display/UI purposes.

4.5.3 Bad Randomness

Definition: Bad randomness refers to the use of insecure or manipulable values—typically derived from blockchain state variables—for generating random numbers. Since these values are either publicly known or subject to proposer/miner influence, they cannot guarantee unpredictability or fairness in randomized smart contract logic [34, 36].

```

pragma solidity ^0.4.0;
contract Lottery {
    event GetBet(uint betAmount, uint blockNumber, bool won);
    struct Bet {
        uint betAmount;
        uint blockNumber;
        bool won;
    }
    address private organizer;
    Bet[] private bets;
    function Lottery() { organizer = msg.sender; }
    function() { throw; } // legacy revert
    // Make a bet
    function makeBet() {
        // Won if block number is even (public, miner-influenced; not random)
        // <yes> <report> BAD_RANDOMNESS
        bool won = (block.number % 2) == 0; // (Line 17) Vulnerability: predictable "randomness"
        // Record the bet with an event
        // <yes> <report> BAD_RANDOMNESS
        bets.push(Bet(msg.value, block.number, won)); // (Line 20) Uses same manipulable source in outcome
        log
    }
}

```

```

        if (won) {
            if (!msg.sender.send(msg.value)) {
                throw;
            }
        }
    }
    function getBets() {
        if (msg.sender != organizer) { throw; }
        for (uint i = 0; i < bets.length; i++) {
            GetBet(bets[i].betAmount, bets[i].blockNumber, bets[i].won);
        }
    }
    function destroy() {
        if (msg.sender != organizer) { throw; }
        suicide(organizer);
    }
}

```

Listing 16. *Bad randomness Vulnerability*

Explanation of Vulnerability:

- **Line 17 – Predictable entropy** The outcome is derived from `block.number % 2`. Block numbers are public and known at inclusion time; proposers/miners can influence which transactions land in even vs. odd blocks.
- **Line 20 – Outcome tied to same block** The contract records `block.number` alongside the bet, cementing a direct link between the manipulable variable and the payout decision in the same transaction.
- **Single-transaction finality** No commit-reveal or delay separates the user’s input from the entropy sample, enabling atomic manipulation or strategic reordering.

Why This is a Vulnerability:

- **Public and miner-influenced inputs:** `block.number`, `block.timestamp`, `blockhash(n)` (for small n), and similar fields are observable and, within limits, biasable by block proposers/validators.
- **Predictability breaks fairness:** Adversaries can forecast outcomes before confirmation and choose when to submit, cancel, or reorder transactions to gain a systematic edge.
- **Atomic exploitation:** Without a delayed reveal or external verifiable randomness, attackers can shape inclusion (e.g., target even blocks) and immediately capture favorable payouts.

Exploitation Scenario:

- **Gas-price frontrun to even blocks:** An attacker observes that even block numbers win. They resubmit their bet with a higher gas price when the mempool suggests the next block will be even, increasing the chance their bet executes in a winning block.
- **Proposer/miner bias:** A block proposer places their own bet transactions only into even-numbered blocks (or withdraws/defers inclusion) to harvest payouts reliably.
- **MEV relay strategy:** An MEV bot sandwiches user bets to ensure its own transaction lands in a favorable parity block while pushing others into losing ones.

Mitigation Recommendations:

- **Use verifiable randomness (preferred):** Integrate a VRF oracle (e.g., on-chain VRF) to obtain an unpredictable, publicly verifiable random value.
- **Commit–reveal with delay:** Require users to commit a hashed secret first and reveal after k blocks. Combine user salt with future, *not yet known* entropy to reduce proposer bias.

- **Separate phases across blocks:** Sample entropy from a block *after* the bet is irreversibly recorded (e.g., use block $n+k$) to limit same-block manipulation; avoid direct use of `block.timestamp/block.number`.
- **Aggregate multi-party entropy:** Mix multiple independent sources (user commits + VRF + operator beacons) so that no single party can unilaterally bias the output.
- **Audit outcome linkage:** Ensure payout logic references only verifiable, unbiased randomness and that logs/events do not encode manipulable “random” fields as authoritative outcomes.

4.6 Access Control and Authentication Vulnerabilities

This family addresses failures in privilege enforcement—when ownership or authorization checks are missing, inconsistent, or incorrectly implemented. Such flaws directly expose administrative or monetary functions to unauthorized users. They represent one of the most common and severe root causes across DeFi incidents.

4.6.1 Access Control Misconfiguration

Definition: An *access control misconfiguration* vulnerability occurs when a smart contract fails to enforce or improperly implements authorization checks on sensitive functions. Such misconfigurations allow unauthorized users to perform privileged actions—such as transferring Ether, modifying critical state variables, or bypassing restrictions—that should be reserved for the contract owner or designated roles [1, 2, 11, 12, 37]. Common causes include missing `require()` statements, incorrect use of ownership patterns, or omission of balance resets and state updates after privileged operations.

```
pragma solidity ^0.4.24;
contract Wallet {
    address creator;
    mapping(address => uint256) balances;

    constructor() public {
        creator = msg.sender;
    }
    function deposit() public payable {
        assert(balances[msg.sender] + msg.value > balances[msg.sender]);
        balances[msg.sender] += msg.value;
    }
    function withdraw(uint256 amount) public {
        require(amount <= balances[msg.sender]);
        msg.sender.transfer(amount);
        balances[msg.sender] -= amount;
    }
    function refund() public {
        // <yes> <report> ACCESS_CONTROL
        // Vulnerability: no authorization or balance reset
        msg.sender.transfer(balances[msg.sender]);
    }
    function migrateTo(address to) public {
        // Only creator can migrate funds
        require(creator == msg.sender);
        to.transfer(this.balance);
    }
}
```

Listing 17. Access control Vulnerability

Explanation of Vulnerability:

- **Lines 18–22 — Unprotected refund path** - The function `refund()` (Line 18) lacks any authorization or role check before transferring funds. The inline marker (Line 19) flags this as `ACCESS_CONTROL`, yet no `require(...)` is present to restrict callers.

- **Line 21 — Missing state update after transfer** - `msg.sender.transfer(balances[msg.sender])` sends Ether but *does not* reset `balances[msg.sender]` to zero. This enables repeated calls to `refund()` to withdraw the same recorded balance multiple times.
- **Missing line (after Line 21) — Absent balance reset** - A critical line such as `balances[msg.sender] = 0;` is missing immediately after the transfer, leaving the per-user credit intact and enabling unlimited refunds.
- **Policy inconsistency (Lines 23–26) vs. refund (Lines 18–22)** - `migrateTo()` correctly enforces `require(creator == msg.sender)` (Line 25), but `refund()` applies no similar check for a fund-moving operation, creating an inconsistent access policy.

Consequently, any caller can repeatedly drain Ether via `refund()` due to missing authorization and missing post-transfer balance reset, leading to complete contract depletion.

Furthermore, although `migrateTo()` correctly restricts access to the contract creator, the absence of similar access validation in `refund()` creates a partial protection gap [11, 12].

Why This is a Vulnerability:

- **Missing state update:** The user's balance is not reset after transferring funds, allowing repeated withdrawals.
- **Improper privilege enforcement:** Any account can call `refund()` without requiring ownership or authorization.
- **Inconsistent access policy:** The contract enforces access control in one function (`migrateTo()`) but neglects it in another that performs critical fund transfers.
- **Financial loss:** This leads to unlimited refunds and complete depletion of the contract's Ether balance [2, 37].

Exploitation Scenario: An attacker deposits a small amount of Ether (e.g., 1 wei) using `deposit()`, which credits their balance. They then repeatedly invoke `refund()` to transfer their full recorded balance each time, without the contract ever resetting it to zero. Each invocation drains an equal amount from the contract's total Ether holdings. Since the function lacks both state mutation and access restrictions, a single attacker can fully exhaust the wallet's balance. This vulnerability was observed in multiple early Ethereum wallets and crowdsale contracts.

Mitigation Recommendations:

- **Reset state after critical actions:** Always update user balances or access flags before or immediately after performing fund transfers:

```
uint256 amount = balances[msg.sender];
balances[msg.sender] = 0;
msg.sender.transfer(amount);
```

- **Use ownership and role-based access control:** Employ the OpenZeppelin Ownable or AccessControl modules to explicitly define administrative privileges:

```
modifier onlyOwner() {
    require(msg.sender == owner, "Not authorized");
}
```

- **Follow the Checks–Effects–Interactions (CEI) pattern:** Verify preconditions, update internal state, and only then interact with external addresses. This prevents reentrancy and ensures correct state before external transfers.
- **Comprehensive testing and auditing:** Use automated analyzers (e.g., Slither, SmartBugs, Mythril) and manual audits to verify consistent access control across all critical functions [11, 12, 37].

- **Consistent design policy:** Define a unified access policy for the contract, ensuring all fund-moving or state-changing functions follow the same authentication and update logic.

Notes: Access control misconfigurations remain one of the most prevalent and high-impact vulnerabilities in smart contracts. Even simple omissions—such as forgetting to reset balances or missing a `require(msg.sender == owner)` statement—can lead to full asset compromise. Modern frameworks and static analyzers include explicit detection heuristics for missing state updates and inconsistent access validation.

4.6.2 `tx.origin` Misuse

Definition: The misuse of `tx.origin` occurs when a contract relies on this global variable to authorize users. Unlike `msg.sender`, which refers to the immediate caller, `tx.origin` refers to the original external account that initiated the transaction. This makes it vulnerable to phishing-style attacks, where a malicious contract tricks a user into triggering sensitive logic in another contract [2, 35].

```
contract TxOriginExample {
    address public owner;
    constructor() {
        owner = msg.sender;
    }
    function transferOwnership(address newOwner) public {
        // Vulnerability: insecure use of tx.origin for authorization
        require(tx.origin == owner, "Not authorized");
        owner = newOwner;
    }
    function safeTransfer(address newOwner) public {
        // Warning: combining msg.sender and tx.origin still unsafe
        require(msg.sender == owner && tx.origin == owner, "Still unsafe pattern");
        owner = newOwner;
    }
}
```

Listing 18. `tx.origin` Vulnerability

Explanation of Vulnerability:

- **Line 8 -** The function `transferOwnership` uses `require(tx.origin == owner)` for access control. If a malicious contract indirectly invokes this function while the victim is the transaction origin, the check passes—despite the actual caller being malicious.
- **Line 13 -** Even though `safeTransfer` uses both `msg.sender` and `tx.origin`, the inclusion of `tx.origin` still makes it unsafe. This line triggers `heuristic_tx_origin_check`.

Why This Is a Vulnerability:

- The use of `tx.origin` exposes the contract to phishing-like attacks. A malicious contract can be designed to execute the vulnerable function while the user unknowingly acts as the transaction origin.
- Since `tx.origin` cannot distinguish between trusted and untrusted intermediate calls, it should never be used in critical logic such as ownership validation.

Exploitation Scenario:

- An attacker deploys a contract that calls `transferOwnership`.
- The victim interacts with this attacker contract. Because the victim's wallet is the transaction origin, the check on Line 9 passes.
- Ownership is silently transferred to the attacker without the victim's knowledge.

Mitigation Recommendations:

- Always use `msg.sender` for authorization checks to validate the direct caller.
- Never use `tx.origin` in ownership or sensitive access logic.
- Use established security libraries such as `Ownable` from OpenZeppelin to handle ownership safely.

4.6.3 Unauthorized Selfdestruct (Suicidal Contract)

Definition: A suicidal contract vulnerability arises when a contract includes a publicly accessible or improperly restricted `selfdestruct` call, allowing unauthorized users to irreversibly terminate the contract and redirect any remaining Ether [11, 22].

```
contract SuicideContract {
    address public owner;
    constructor() {
        owner = msg.sender;
    }
    function destroy() public {
        // Vulnerability: selfdestruct is called without checking ownership
        selfdestruct(payable(msg.sender));
    }
}
```

Listing 19. Unprotected Selfdestruct Vulnerability

Explanation of the Vulnerability:

- **Line 8 – Selfdestruct without access control -** The `destroy()` function is publicly accessible and allows any caller to execute `selfdestruct`. It transfers the entire contract balance to the caller (`msg.sender`) without verifying if the caller is the owner. This violates fundamental access control expectations for critical operations.

Why This Is a Vulnerability:

- The contract can be terminated at any time by any address.
- Ether stored in the contract is transferred to the attacker.
- Contract functionality is destroyed permanently, disrupting dependent applications and users.

Exploitation Scenario:

- An attacker discovers the `destroy()` function and calls it.
- The contract is terminated, and any remaining funds are sent to the attacker.
- All future calls to the contract fail, since the bytecode is removed from the blockchain.

Mitigation Recommendations:

- Enforce strict access control before calling `selfdestruct`. For example:

```
require(msg.sender == owner, "Not authorized");
```

- Use the `onlyOwner` modifier or similar role-based access control for destructive functions.
- Avoid using `selfdestruct` unless contract destruction is part of the design. Prefer disabling functionality via flags.

4.7 ABI and Input Validation Vulnerabilities

At the interface between users and contract logic, improper validation of calldata or encoded parameters can shift argument boundaries or allow ambiguous interpretation. These vulnerabilities manifest as short-address bugs, unsafe `bytes` decoding, or hash collisions caused by packed encodings.

4.7.1 Short Address Attack

Definition: Short Address Attack exploits Ethereum's padding and alignment behavior for function arguments, particularly when address-type inputs are passed with fewer than 20 bytes, causing subsequent parameters to be misinterpreted [12].

```
pragma solidity ^0.4.24;
contract ShortAddressAttackExamples {
    mapping(address => uint256) public balances;
    // Classic short address vulnerability using raw address
    function transfer(address to, uint256 amount) public {
        balances[to] += amount; // Vulnerability: No input length check for 'to' (short address attack)
        balances[msg.sender] -= amount;
    }
    // Uses bytes input assumed to contain an address
    function unsafeSetAddress(bytes memory raw) public {
        address extracted;
        assembly {
            extracted := mload(add(raw, 20)) // Vulnerability: Unsafe use of mload on unchecked bytes
        }
        balances[extracted] = 100;
    }
    // Uses an address input in a call without checking length
    function callReceiver(address receiver) public {
        receiver.call("");
    }
    // Safe version with length check
    function safe(bytes memory raw) public {
        require(raw.length == 20, "Invalid length");
        address extracted;
        assembly {
            extracted := mload(add(raw, 20))
        }
        balances[extracted] = 123;
    }
}
```

Listing 20. Short Address Vulnerability

Explanation of Vulnerabilities:

- **Line 6 – Raw address parameter without length check** The `transfer` function uses the `to` address parameter directly. When encoded calldata is too short, Solidity may misalign the address and `uint256` parameters, causing fund misdirection.
- **Line 13 – Unsafe assembly with unchecked bytes** The function assumes `raw` contains at least 20 bytes but does not check this before accessing memory directly via `mload`. This introduces undefined behavior and may extract incorrect addresses.
- **Line 19 – External call to unchecked address** The `receiver` parameter is used in a low-level call without confirming whether the address is properly constructed or safe.
- **Line 22 – Secure implementation with length validation** Serves as a best practice: explicitly checks that `raw.length == 20` to ensure integrity before extracting an address.

Why This Is a Vulnerability:

- Ethereum pads calldata on the right, and improper input length causes offsets to shift unexpectedly.
- Attackers can exploit this by crafting calldata that causes value misalignment or function argument corruption.
- Inline assembly makes the attack surface worse if bytes are not checked before memory reads.

Exploitation Scenario:

- An attacker sends a 19-byte address followed by carefully crafted values, causing the address and amount parameters in `transfer` to be interpreted incorrectly.
- This can result in tokens being sent to the wrong address or balance underflows that favor the attacker.
- If unchecked bytes are passed to `unsafeSetAddress`, it may silently write balance to unintended addresses.

Mitigation Recommendations:

- Always validate the length of `bytes` before extracting addresses from them:

```
require(raw.length == 20, "Invalid address length");
```

- Avoid direct memory reads using inline assembly unless absolutely required.
- Upgrade to modern Solidity versions ($\geq 0.5.0$) which enforce stricter input checking.
- Use well-tested libraries for encoding/decoding calldata or low-level operations.

4.7.2 Packed-ABI Hash Collision

Definition: When hashing concatenated variable-length fields via `abi.encodePacked`, distinct tuples can serialize to identical byte streams, yielding the same hash and enabling intent confusion or signature/commit forgery [2,3].

```
pragma solidity ^0.8.0;
contract PackedABICollision {
    mapping(bytes32 => bool) public executed;
    // Vulnerable: variable-length fields packed without separators/types
    function authorize(string memory action, string memory param) external {
        // Collisions: ("ab", "c") vs ("a", "bc") -> same packed bytes
        // <yes> <report> PACKED_ABI_COLLISION
        bytes32 h = keccak256(abi.encodePacked(action, param)); // Vulnerability: ambiguous packing
        require(!executed[h], "already used");
        executed[h] = true;
        // ... perform action "authorized" by (action,param)
    }
    // Safer alternative:
    function authorizeTyped(string memory action, string memory param) external {
        // abi.encode is typed/length-prefixed -> no such collisions
        bytes32 h = keccak256(abi.encode(action, param));
        require(!executed[h], "already used");
        executed[h] = true;
    }
}
```

Listing 21. Packed-ABI Hash Collision Vulnerability

Explanation of Vulnerabilities

- **Line 8 — Ambiguous concatenation** `abi.encodePacked(action,param)` elides type and length prefixes; variable-length strings can produce identical concatenations, e.g., ("ab", "c") and ("a", "bc") serialize to the same bytes.
- **Line 8 — Colliding hash as authority key** The contract derives `h` from the ambiguous packed bytes and treats it as an authorization/uniqueness key, so different input tuples can map to the same `h`.
- **Lines 8–9 — Logic bound to h** Both the replay check and state update hinge solely on `h`; a collision lets an attacker pass the check with semantically different parameters or pre-burn a victim's hash.
- **Scope:** The issue generalizes beyond strings to any mix of variable-length fields (e.g., `bytes`, `string`) and even some fixed-width concatenations when not delimited with domain separators.

Why This is a Vulnerability:

- **Input ambiguity undermines binding:** The hash no longer uniquely binds the intended tuple (`action`, `param`); distinct inputs can authorize the same effect.
- **Second-preimage feasibility:** Finding an alternate tuple that packs to an existing byte stream is trivial (string splicing, crafted `bytes`), enabling commit forgery and signature misuse.
- **State/key collisions:** Using the colliding `h` as a mapping key allows preemption (burning) or impersonation of another tuple's authorization.
- **Cross-domain confusion:** Reusing packed hashes across subsystems (orders, roles, claims) without domain separation enables unintended replay or privilege escalation.

Exploitation Scenario:

- **Parameter substitution:** A victim intends to authorize ("ab", "c"). The attacker submits ("a", "bc"), which produces the same `h` (Line 7), passing the uniqueness check (Line 8) and executing unintended logic.
- **Pre-burn (DoS):** The attacker first calls `authorize("a", "bc")` to set `executed[h]=true` (Line 9), preventing the victim's later, legitimate `authorize("ab", "c")`.
- **Signature replay across intents:** If an off-chain signature covers `keccak256(abi.encodePacked(action, param))`, the attacker replaces the signed tuple with a colliding pair and reuses the signature to authorize a different action.

Mitigation Recommendations:

- **Use typed encoding:** Prefer `keccak256(abi.encode(...))` so type and length prefixes remove ambiguity (see `authorizeTyped`).
- **Add domain separation:** Prefix encodings with a constant tag and version (e.g., `keccak256(abi.encode("AUTH_V1", action, param))`).
- **Avoid packing multiple dynamic fields:** If `abi.encodePacked` must be used, do not concatenate two or more dynamic types; alternatively, include explicit length delimiters for each dynamic field.
- **Ethereum Improvement Proposal (EIP)-712 for signatures:** Use typed structured data with domain separator and explicit field types when producing/verifying off-chain signatures.
- **Hash fields individually then combine:** For dynamic fields, hash each separately and then encode the fixed-size digests: `keccak256(abi.encode(keccak256(bytes(action)), keccak256(bytes(param))))`.
- **Property tests and fuzzing:** Add invariants (no two distinct tuples yield the same key) and fuzz over adversarial string/`bytes` splits to catch collisions in tests.

Use `abi.encode` for preimages, or include explicit separators/lengths. Prefer EIP-712 typed data for signatures to bind types and domain [2, 3].

4.8 Network and Protocol Layer Vulnerabilities

Beyond individual contracts, cross-domain protocols such as bridges, relayers, and signature schemes introduce systemic risks when message semantics are not cryptographically bound to their origin. This family captures replay, double-mint, and finality-mismatch vulnerabilities that arise from missing domain separation, nonces, or confirmation depth.

4.8.1 Replay Attack

Definition: A *replay attack* occurs when a valid transaction or signed message intended for one context (e.g., a particular chain, domain, or session) is maliciously or inadvertently re-submitted in a different context where it has undesired effects. In smart-contract systems this commonly manifests as (a) signature replay across chains or domains, and (b) re-execution of messaging/bridge instructions due to missing nonces or domain separation [1,4,5].

```
pragma solidity ^0.6.0;
contract SimpleRelayer {
    address public owner;
    constructor(address _owner) public { owner = _owner; }
    // Vulnerable: verifies a signature without domain/nonce/expiry
    function execute(bytes32 payload, bytes memory sig) public {
        // msgHash = keccak256(payload) -- no chainId, no nonce, no deadline
        // <yes> <report> REPLAY_ATTACK
        bytes32 msgHash = keccak256(abi.encodePacked(payload)); // (Line 8) missing domain/nonce/expiry
        address signer = recoverSigner(msgHash, sig); // (Line 9) recovers over weak digest
        require(signer == owner, "not authorized"); // (Line 10) no replay guard or scoping
        // perform privileged action tied only to payload
        _doAction(payload); // (Line 11) effect re-executable elsewhere
    }
    function _doAction(bytes32) internal {
        // privileged behavior...
    }
    function recoverSigner(bytes32 h, bytes memory sig) internal pure returns (address) {
        // standard ecrecover wrapper (omitted)
    }
}
```

Listing 22. Replay Attack Vulnerability

Explanation of Vulnerability:

- **Lines 8 — Digest lacks domain separation** The signed hash is computed over payload alone via `keccak256(abi.encodePacked(payload))`, omitting chain id, contract address, and purpose. Identical bytes validate across contracts/chains.
- **Line 8 — No nonce / single-use token** The digest contains no per-user or per-message nonce, so the same signature can be submitted repeatedly (same chain) or at a later time.
- **Line 9 — Authorization over weak message** `recoverSigner` authenticates the signer but not the *context* of execution; recovering over an ambiguous digest makes cross-domain reuse trivial.
- **Line 10 — Missing replay guard** No used[signer][nonce] or consumed[msgHash] mapping is checked/updated; the message remains valid indefinitely.
- **Line 11 — Privileged effect tied only to payload** The privileged action binds to payload without any consumed-ticket semantics, enabling the same logical instruction to be re-applied.

Why This is a Vulnerability:

- **Cross-context validity:** A signature over a bare payload is valid wherever identical bytes are accepted (other chains, mirror contracts, replay via relayers), so intent cannot be constrained.
- **Idempotence / re-use:** Without single-use nonces or deadlines, attackers can replay the same authorization to repeat the effect.
- **Bridge and cross-domain risk:** Relayers/bridges that omit replay filters allow previously processed instructions to be executed again on the destination chain [4,5].
- **High impact:** Leads to double-spends, duplicate withdrawals/mints, or replay of administrative operations.

Exploitation Scenario:

- **Cross-chain signature replay:** The owner signs “withdraw 100” intended for Chain A. An adversary replays the same bytes against an equivalent contract on Chain B; recovery succeeds and funds are unjustly withdrawn on B.
- **Same-chain multi-use replay:** The attacker resubmits the same signed payload multiple times to repeat a privileged action because no nonce was consumed on first execution.
- **Bridge message re-execution:** A previously processed bridge message is re-submitted to the destination contract; lacking consumed-message tracking, the contract mints a second time.

Mitigation Recommendations:

- **Domain separation (EIP-712 / EIP-191):** Sign structured data bound to a domain (name, version, chainId, verifyingContract). Verify on-chain against the same domain.

```
// Pseudocode: EIP-712 digest = keccak256(
// "\x19\x01", DOMAIN_SEPARATOR, keccak256(encodeTyped(struct)))
```

- **Nonces and single-use tokens:** Include a per-user/per-message nonce and record consumption.

```
mapping(address => mapping(uint256 => bool)) public usedNonce;
require(!usedNonce[signer][nonce], "replay");
usedNonce[signer][nonce] = true;
```

- **Expiration windows:** Require deadline >= block.timestamp in the signed data; reject stale messages.
- **Chain-id binding:** Include chainId in the signed content (via EIP-712 domain or explicit field) so signatures are invalid on other chains.
- **Bridge idempotency:** Track and reject re-processing of messageId/txHash on destination chains; persist proofs/nonces to enforce one-time execution.
- **Hardened libraries:** Use well-audited EIP-712 helpers and signature-verification libraries that enforce domain, nonce, and deadline checks by default.
- **Testing & analysis:** Add tests for cross-chain/same-chain replay; use static/dynamic analyzers to flag verification code lacking domain, nonce, or expiry [4,5].

Notes: Replay protection is foundational for cross-domain security; recent surveys and bridge incident analyses emphasize domain separation, nonces, deadlines, and explicit message-IDs as standard defenses against replay attacks [1,4,5].

4.8.2 Cross-Chain Finality/Reorg Exploit (bridge double-mint)

Definition: If a bridge honors source-chain events before they are final (or with weak/optimistic proofs), a later reorg can remove the source Lock while the destination Mint remains, producing unbacked assets (double-mint) [4,5].

```
/*
 * Pseudocode (destination chain):
 */
contract LazyBridge {
    mapping(bytes32 => bool) public processed;

    // Vulnerable: accepts "lock" events with weak/optimistic proof and shallow confs
    function redeem(bytes32 eventId, bytes calldata lightProof, address to, uint256 amt) external {
        require(verifyLightProof(lightProof, eventId), "invalid proof"); // (Line 9) weak proof acceptance
        require(!processed[eventId], "already processed"); // (Line 10) replay guard local only
        processed[eventId] = true;
        _mint(to, amt); // (Line 12) Vulnerability: mint before deep finality/dispute window
    }

    function verifyLightProof(bytes calldata, bytes32) internal pure returns (bool) {
```

```

        // returns true for brevity; real impl would check validator sigs, etc. // (Line 16) trivial proof
        stub
    }
    return true;
}

function _mint(address, uint256) internal { /* mint on destination */ }
}

```

Listing 23. *Cross-Chain Finality/Reorg Exploit Vulnerability*

Explanation of Vulnerability:

- **Lines 8–12** — Mint on optimistic proof `redeem()` accepts a light/optimistic proof and immediately mints, without waiting for deep confirmations, finalized headers, or a dispute window.
- **Line 10** — Local-only replay guard `processed[eventId]` prevents re-processing on the destination chain but does *not* tie the mint’s validity to the source event’s persistence.
- **Line 16** — Insufficient verification `verifyLightProof` is a placeholder for shallow checks; in practice, weak committee signatures or short confirmations are not robust against reorganizations or equivocation.
- **Protocol boundary:** Asynchronous consensus across chains means history can diverge temporarily; without finality-aware proofs, a destination mint may outlive a pruned source lock.

Why This is a Vulnerability:

- **Finality mismatch:** Destination state is updated (mint) on evidence that can be revoked on the source (reorg), creating unbacked liabilities.
- **Asymmetric reversibility:** Source Lock can disappear after a reorg, while destination Mint is irreversible (no automatic unwind), leading to permanent supply inflation.
- **Weak trust base:** Light/optimistic proofs with shallow confirmations or small validator quorums are vulnerable to short-range reorgs and byzantine signer sets.
- **High impact domain:** Bridges custody or synthesize large asset values; a single exploit yields systemic losses (double-mint, under-collateralization, insolvency).

Exploitation Scenario:

- **Timed lock + instant redeem:** The attacker locks on Chain A at shallow depth, then immediately submits a light proof to Chain B and receives a mint.
- **Induced reorg / equivocation:** The attacker (or colluding proposers) triggers a short-range reorg on Chain A that removes the lock transaction. Chain B retains the minted tokens.
- **Exit with profit:** The attacker trades/bridges out the unbacked tokens on Chain B before any manual reconciliation, realizing profit while the system is under-collateralized.

Mitigation Recommendations:

- **Finality-aware proofs:** Require finalized headers (e.g., Byzantine Fault Tolerance (BFT) finality, checkpointed epochs) or zk/light-client verification of inclusion against finalized states before minting.
- **Dispute / challenge windows:** Use optimistic proofs only with a challenge period; defer mint or mint-to-escrow until the window elapses without a valid fraud proof.
- **Depth thresholds and reorg handling:** Enforce conservative confirmation depths on probabilistic-finality chains and implement automatic *rollback* or *clawback* procedures if the source event is later orphaned.

- **Two-phase settle:** Mint to an escrowed/“pending” balance first; release to the user only after finality is attained, or allow slashable relayers to underwrite early release.
- **Cross-chain replay/uniqueness:** Bind `eventId` to source chain id, height, and Merkle/accumulator root; store a cryptographic commitment so that removed events cannot be proven later.
- **Economic security:** Require bonded relayers/guardians with slashing for incorrect attestations; size bonds relative to TVL and potential reorg depth.
- **Rate limits and circuit breakers:** Cap per-block/per-epoch mints and enable pausability on proof anomalies or reorg detection.
- **Monitoring and audits:** Continuously monitor reorg depth on source chains; formally verify proof-verification logic and finalize/rollback state machines.

Practical Mapping and Tool Alignment. To bridge the conceptual taxonomy of Section 3 with concrete detection practice, Table 1 summarizes how each vulnerability family manifests at the program-analysis level. It lists representative *detection signals* commonly used by static, dynamic, and learning-based tools, and indicates which major datasets or surveys include examples of each category. This mapping helps researchers and auditors quickly align vulnerability families with existing evaluation benchmarks and detection frameworks, facilitating unified reporting and tool comparison.

TABLE 1. Consolidated taxonomy of smart-contract vulnerability families with representative detection signals and datasets/tools.

Family	Detection signals (typical)	Datasets / Tools / Surveys
Control-Flow Interaction	External call before state update; recursive entry; ordering/race on shared state	SmartBugs [11]; SolidiFI [12]; tool surveys [2,37]
External Call & Error Handling	Unchecked <code>call/send/transfer</code> ; swallowed revert; push-payment anti-pattern	SmartBugs [11]; SolidiFI [12]; surveys [2,37]
State / Storage Integrity	Untrusted <code>delegatecall</code> ; proxy target drift; slot overlap; unsafe Yul writes	Upgrade/storage analyses [7, 8]; taxonomies [2,4]
Arithmetic & Type Safety	Overflow/underflow (pre-0.8); div-by-zero; narrow casts	SmartBugs [11]; classic surveys [1, 12,37]
Environment Dependency	Use of <code>block.timestamp/number/hash</code> for control or “randomness”	Randomness/miner-bias studies [34,36]; surveys [2]
Access Control & Authentication	Missing <code>require</code> /modifiers; <code>tx.origin</code> ; unprotected <code>selfdestruct</code>	Core surveys [1,2]; selfdestruct handling [22]
ABI / Input Validation	Short-address; packed-ABI collisions; unchecked <code>bytes</code> decoding; no EIP-712	Packed-ABI [2, 3]; datasets/surveys [12,37]
Network / Protocol Layer	Domainless signatures; missing nonces/expiry; bridge finality/re-org and replay	Bridge/cross-chain [4,5]; classic replay [1]

5 Discussion

Consolidation of perspectives. Our eight-family taxonomy unifies diverse vulnerability lists and ad-hoc labels under four recurring structural dimensions: (i) control-flow and execution

ordering, (ii) external-call and exception handling, (iii) state integrity and privilege boundaries, and (iv) environmental or cross-domain assumptions. By focusing on these first-cause axes, the framework directly connects root defects with mitigation strategies—for example, Checks–Effects–Interactions (CEI) for reentrancy and transaction-ordering flaws, or domain separation for cross-chain replay prevention.

Implications for detection and benchmarking. Instead of counting isolated bug names, tool evaluations can now report coverage along structural axes. This perspective exposes where static, dynamic, or ML-based detectors overlap or diverge and clarifies which analysis signals (e.g., control-flow slicing, privilege validation, or ABI decoding) each approach captures. Relabeling benchmark datasets such as SmartBugs and SolidiFI according to these unified families can reduce naming drift and prevent double-counting of near-synonymous issues. In practice, auditors gain a clearer mapping between reported findings and root-cause categories, while tool builders obtain standardized coverage metrics.

Explainability and multi-axis incidents. Some exploits cross multiple structural boundaries—for example, ERC-20 approval races mix ordering and authorization faults. Following our *first-cause principle*, each case is assigned to the dimension that best explains the exploit with minimal auxiliary assumptions. This rule improves consistency in dataset labeling and in human-interpretable tool output, where each alert can reference the triggering lines, violated principle, and a minimal counterexample trace.

Limitations and scope. The taxonomy currently targets EVM-compatible ecosystems; porting to non-EVM or BFT-based chains may require adjusting environmental and cross-domain categories. Machine-learning models trained on legacy labels will require re-annotation to align with the unified schema—a one-time adaptation that yields more consistent evaluation and clearer coverage reporting. Overall, the framework provides a reusable lens for researchers, educators, and auditors to communicate about vulnerabilities with shared terminology.

6 Legacy and Historical Attack Terminology

Motivation. Terminology describing smart-contract vulnerabilities has evolved through incident reports, community discussions, and early static-analysis tools. Labels such as “gasless send,” “short address,” or “call to unknown” were often coined ad-hoc, capturing symptoms rather than structural causes. Although subsequent taxonomies (e.g., Decentralized Application Security Project (DAPS), SWC, and academic surveys [1–5]) brought partial standardization, inconsistent naming still hampers comparison across tools and benchmarks.

Mapping rationale. We align these legacy terms with our root-cause-oriented taxonomy from Section 3, following the *first-cause principle*—each legacy label is mapped to the structural dimension that most directly explains its exploit mechanics. This mapping clarifies equivalences between historical and modern terminologies, enabling backward compatibility for datasets and reproducible evaluation.

Mapping table. Table 2 catalogs widely used legacy vulnerability names alongside their historical or alternative terminology found in prior research, tools, and practitioner discourse. These include both formal terms coined in academic papers and informal labels popularized through audit reports or community lore. The mapping serves two key purposes: (1) to enable interpretability and compatibility with datasets or tools that still use legacy labels, and (2) to highlight how terminology has evolved and converged toward structured, root-cause-based classifications.

TABLE 2. Legacy vulnerability terms and their alternative or historical counterparts.

Legacy Term	Alternative / Historical Term(s)
Reentrancy	Recursive Call, Reentrant Call
Transaction Ordering Dependence (TOD)	Front-running, Race Condition
Checks–Effects–Interactions Violation	Call Before State Change, Unsafe Call Order
Delegatecall Injection	Context Confusion, Proxy Takeover, Logic Hijack
Unprotected Selfdestruct	Suicidal Contract, Accidental Destruction
<code>tx.origin</code> Misuse	Phishing via <code>tx.origin</code> , Origin Confusion
Short Address Attack	Calldata Truncation Bug, ABI Padding Mismatch
Packed-ABI Hash Collision	Hash Collision in Commitments, Ambiguous Encoding
Unchecked Low-Level Calls	Ignored Return Values, Fault-oblivious Call
On-Chain Randomness Misuse	Miner-Manipulated Randomness, Block Entropy Exploit
Approval Race Condition	ERC20 Allowance Double-Spend, Approve-/TransferFrom Race
Access Control Misconfiguration	Missing <code>onlyOwner</code> , Weak Privilege Checks
Bridge Replay / Finality Exploit	Double Mint, Cross-Chain Reorg Attack
Signature Replay Attack	Domainless Signature Reuse, Nonce-less Authorization
Integer Overflow/Underflow	Arithmetic Bug, Wraparound Error
Division by Zero	Math Error, Runtime Arithmetic Exception
Unchecked Assembly Decoding	Raw Memory Access, Unsafe Byte Parsing
DoS via Revert or Gas Exhaustion	Griefing Attack, Fallback Denial

7 Conclusion

This study presented an eight-family taxonomy that organizes Ethereum smart-contract vulnerabilities by their program-structural root causes rather than by inconsistent historical names. Each family was illustrated through representative code examples, exploit mechanics, and practical mitigations, revealing how diverse incidents converge on a small set of recurring design flaws. By aligning terminology, detection signals, and datasets, the taxonomy improves cross-tool comparability and fosters explainable, root-cause-oriented analysis.

Practical impact. For auditors and developers, the taxonomy serves as a diagnostic checklist that links specific bug patterns to their underlying structural causes and canonical defenses. For tool builders and researchers, it defines common evaluation axes, simplifying coverage measurement and dataset harmonization. For educators, it offers a coherent structure for teaching smart-contract security without relying on legacy, symptom-based labels.

Future work.

Future directions include: (i) expanding the taxonomy to cover economic-layer and oracle manipulation patterns; (ii) formalizing bridge and layer-2 dispute mechanisms; (iii) releasing a re-labeled benchmark with per-axis coverage metrics and explainability artifacts; and (iv) evaluating LLM-assisted detectors that output structured rationales aligned with our taxonomy.

Collectively, these efforts aim to move the ecosystem from fragmented naming toward unified, interpretable assurance.

References

- [1] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (sok),” in *International conference on principles of security and trust*. Springer, 2017, pp. 164–186.
- [2] H. Chu, P. Zhang, H. Dong, Y. Xiao, S. Ji, and W. Li, “A survey on smart contract vulnerabilities: Data sources, detection and repair,” *Information and Software Technology*, vol. 159, p. 107221, 2023.
- [3] F. R. Vidal, N. Ivaki, and N. Laranjeiro, “Vulnerability detection techniques for smart contracts: A systematic literature review,” *Journal of Systems and Software*, p. 112160, 2024.
- [4] T. Jiao, Z. Xu, M. Qi, S. Wen, Y. Xiang, and G. Nan, “A survey of ethereum smart contract security: Attacks and detection,” *Distributed Ledger Technologies: Research and Practice*, vol. 3, no. 3, pp. 1–28, 2024.
- [5] M. Kezadri Hamiaz and M. Driss, “Ethereum smart contracts under scrutiny: A survey of security verification tools, techniques, and challenges,” *Computers*, vol. 14, no. 6, p. 226, 2025.
- [6] H. Zhu, L. Yang, L. Wang, and V. S. Sheng, “A survey on security analysis methods of smart contracts,” *IEEE Transactions on Services Computing*, 2024.
- [7] G. Iuliano and D. Di Nucci, “Smart contract vulnerabilities, tools, and benchmarks: An updated systematic literature review,” *arXiv preprint arXiv:2412.01719*, 2024.
- [8] M. Pasqua, A. Benini, F. Contro, M. Crosara, M. Dalla Preda, and M. Ceccato, “Enhancing ethereum smart-contracts static analysis by computing a precise control-flow graph of ethereum bytecode,” *Journal of Systems and Software*, vol. 200, p. 111653, 2023.
- [9] Z. Liu, M. Jiang, S. Zhang, J. Zhang, and Y. Liu, “A smart contract vulnerability detection mechanism based on deep learning and expert rules,” *IEEE Access*, vol. 11, pp. 77 990–77 999, 2023.
- [10] S. Wang and X. Zhao, “Contractsentry: a static analysis tool for smart contract vulnerability detection,” *Automated Software Engineering*, vol. 32, no. 1, p. 1, 2025.
- [11] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, “Smartbugs: A framework to analyze solidity smart contracts,” in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 2020, pp. 1349–1352.
- [12] A. Ghaleb and K. Pattabiraman, “How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection,” in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 415–427.
- [13] X. Tang, Y. Du, A. Lai, Z. Zhang, and L. Shi, “Deep learning-based solution for smart contract vulnerabilities detection,” *Scientific Reports*, vol. 13, no. 1, p. 20106, 2023.
- [14] J. Cai, B. Li, J. Zhang, X. Sun, and B. Chen, “Combine sliced joint graph with graph neural networks for smart contract vulnerability detection,” *Journal of Systems and Software*, vol. 195, p. 111550, 2023.
- [15] K. Gong, X. Song, N. Wang, C. Wang, and H. Zhu, “Scgformer: Smart contract vulnerability detection based on control flow graph and transformer,” *IET Blockchain*, vol. 3, no. 4, pp. 213–221, 2023.
- [16] H. Zhang, W. Zhang, Y. Feng, and Y. Liu, “Svscanner: Detecting smart contract vulnerabilities via deep semantic extraction,” *Journal of Information Security and Applications*, vol. 75, p. 103484, 2023.
- [17] J. Li, G. Lu, Y. Gao, and F. Gao, “A smart contract vulnerability detection method based on multimodal feature fusion and deep learning,” *Mathematics*, vol. 11, no. 23, p. 4823, 2023.
- [18] D. Yuan, X. Wang, Y. Li, and T. Zhang, “Optimizing smart contract vulnerability detection via multi-modality code and entropy embedding,” *Journal of Systems and Software*, vol. 202, p. 111699, 2023.
- [19] S.-J. Hwang, S.-H. Choi, J. Shin, and Y.-H. Choi, “Codenet: Code-targeted convolutional neural network architecture for smart contract vulnerability detection,” *IEEE Access*, vol. 10, pp. 32 595–32 607, 2022.
- [20] J. Huang, K. Zhou, A. Xiong, and D. Li, “Smart contract vulnerability detection model based on multi-task learning,” *Sensors*, vol. 22, no. 5, p. 1829, 2022.
- [21] W. Deng, H. Wei, T. Huang, C. Cao, Y. Peng, and X. Hu, “Smart contract vulnerability detection based on deep learning and multimodal decision fusion,” *Sensors*, vol. 23, no. 16, p. 7246, 2023.
- [22] L. S. H. Colin, P. M. Mohan, J. Pan, and P. L. K. Keong, “An integrated smart contract vulnerability detection tool using multi-layer perceptron on real-time solidity smart contracts,” *IEEE Access*, vol. 12, pp. 23 549–23 567, 2024.
- [23] J. Fan, Y. He, and H. Wu, “Small sample smart contract vulnerability detection method based on multi-layer feature fusion,” *Complex & Intelligent Systems*, vol. 11, no. 4, p. 198, 2025.

- [24] C. Y. Lin, H. Zhao, and J. H. Liu, “A lightweight vulnerability detection method for long smart contracts based on bimodal feature fusion,” *Cybersecurity*, vol. 8, no. 1, pp. 1–20, 2025.
- [25] B. Boi, C. Esposito, and S. Lee, “Smart contract vulnerability detection: The role of large language model (llm),” *ACM SIGAPP Applied Computing Review*, vol. 24, no. 2, pp. 19–29, 2024.
- [26] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, J. Yu, Y. Wang, X. Lin, T. Chen, and Z. Zheng, “When chatgpt meets smart contract vulnerability detection: How far are we?” *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–30, 2025.
- [27] H. Ding, Y. Liu, X. Piao, H. Song, and Z. Ji, “Smartguard: An llm-enhanced framework for smart contract vulnerability detection,” *Expert Systems with Applications*, vol. 269, p. 126479, 2025.
- [28] R. Otoni, M. Marescotti, L. Alt, P. Eugster, A. Hyvärinen, and N. Sharygina, “A solicitous approach to smart contract verification,” *ACM Transactions on Privacy and Security*, vol. 26, no. 2, pp. 1–28, 2023.
- [29] H. Wu, Y. Peng, Y. He, and J. Fan, “A review of deep learning-based vulnerability detection tools for ethernet smart contracts.” *Computer Modeling in Engineering & Sciences (CMES)*, vol. 140, no. 1, 2024.
- [30] I. Garfatta, K. Klai, W. Gaaloul, and M. Graiet, “A survey on formal verification for solidity smart contracts,” in *Proceedings of the 2021 Australasian Computer Science Week Multiconference*, 2021, pp. 1–10.
- [31] L. Olivieri and F. Spoto, “Software verification challenges in the blockchain ecosystem,” *International Journal on Software Tools for Technology Transfer*, vol. 26, no. 4, pp. 431–444, 2024.
- [32] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, “Combining graph neural networks with expert knowledge for smart contract vulnerability detection,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 2, pp. 1296–1310, 2021.
- [33] C. Ma, S. Liu, and G. Xu, “Hgat: smart contract vulnerability detection method based on hierarchical graph attention network,” *Journal of Cloud Computing*, vol. 12, no. 1, p. 93, 2023.
- [34] X. Sun, L. Tu, J. Zhang, J. Cai, B. Li, and Y. Wang, “Assbert: Active and semi-supervised bert for smart contract vulnerability detection,” *Journal of Information Security and Applications*, vol. 73, p. 103423, 2023.
- [35] L. Zhang, W. Chen, W. Wang, Z. Jin, C. Zhao, Z. Cai, and H. Chen, “Cbgru: A detection method of smart contract vulnerability based on a hybrid model,” *Sensors*, vol. 22, no. 9, p. 3577, 2022.
- [36] P. Qian, J. He, L. Lu, S. Wu, Z. Lu, L. Wu, Y. Zhou, and Q. He, “Demystifying random number in ethereum smart contract: Taxonomy, vulnerability identification, and attack detection,” *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3793–3810, 2023.
- [37] D. He, R. Wu, X. Li, S. Chan, and M. Guizani, “Detection of vulnerabilities of blockchain smart contracts,” *IEEE Internet of Things Journal*, vol. 10, no. 14, pp. 12 178–12 185, 2023.
- [38] X. Ren, Y. Wu, J. Li, D. Hao, and M. Alam, “Smart contract vulnerability detection based on a semantic code structure and a self-designed neural network,” *Computers and Electrical Engineering*, vol. 109, p. 108766, 2023.