

SoK: Root Causes of \$1 Billion Loss in Smart Contract Real-World Attacks via a Systematic Literature Review of Vulnerabilities

Hadi Rezaei*
 University of Salerno
 hrezaei@unisa.it

Mojtaba Eshghie*
 KTH Royal Institute of Technology
 eshghie@kth.se

Karl Andersson
 Luleå University of Technology
 karl.andersson@ltu.se

Francesco Palmieri
 University of Salerno
 fpalmieri@unisa.it

Abstract—While catastrophic attacks on Ethereum persist, vulnerability research remains fixated on implementation-level smart contract bugs, creating a gap between academic understanding of vulnerabilities and the true root causes of high-impact, real-world incidents. To address this, we employ a two-pronged methodology: first, a systematic literature review of 71 academic papers to build a catalog of 24 active and 5 deprecated vulnerabilities. Second, we conduct an in-depth, empirical analysis of 50 of the most severe real-world attacks between 2022 and 2025, collectively incurring over \$1.09B in losses, to identify their root causes. We introduce the concept of “exploit chains” by revealing that many incidents are not caused by isolated vulnerabilities but by combinations of human, operational, and economic design flaws that link with implementation bugs to enable an attack. Our analysis yields insights on how decentralized applications are exploited in practice, leading to a novel, four-tier root-cause framework that moves beyond code-level vulnerabilities. We find that real-world successful attacks on Ethereum (and related networks) trace back to one of the four tiers of (1) protocol logic design, (2) lifecycle and governance, (3) external dependencies, and (4) classic smart contract vulnerabilities. We investigate the suitability of this multi-tier incident root-cause framework via a case study.

I. INTRODUCTION

The Ethereum blockchain and its smart contract ecosystem have grown to manage hundreds of billions of dollars [1], [2]. Along with this growth, in recent years, a relentless series of successful high-profile hacks has occurred, resulting in staggering financial losses [3], [4]. Some reports suggest up to \$2.2 billion stolen from protocols in 2024 [5]. In response to such attacks, a substantial body of research has been performed to classify and detect smart contract vulnerabilities. These efforts focused on implementation-level bugs such as reentrancy and integer overflows [6], [7]. This led to the development of numerous static and dynamic analysis tools

TABLE I: Comparison of smart contract security SLRs and SoKs.

Paper	Year	Vulns	Prev. ^b	Tax. ^a	Wild ^c	Key Strength
S1 [12]	2020	40	P	4L	X	Vulnerability ↔ attack mapping
S2 [13]	2021	20	X	3C	X	Comprehensive tool analysis
S3 [14]	2021	8	X	N1	C	Maps vulnerabilities → attacks → detection tools
S4 [15]	2021	202	✓	FM	X	Comprehensive formal methods for all SC ⁱ verification.
S5 [16]	2022	6	X	DT	X	Vulnerability detection tools quality gaps
S6 [17]	2022	24	X	RS	X	Causal hierarchy enabling systematic prevention
S7 [18]	2023	12	X	3L	X	Layer-based classification: Solidity, EVM, Blockchain
S8 [19]	2023	6	X	TB	X	Categorizes 6 vulns: (BE/R/C/EX/ED/BD) ^j + tool assessment
S9* [3]	2023	50+	✓	5L	181	Analyzes 181 exploits across NET/CONSC/PRO/AUX ^e layers
S10 [20]	2024	94	X	HR	E	Open hierarchical taxonomy with code examples & ODC ^f :
S11* [21]	2024	V ^d	X	UN	X	Numbers=exploit count, C=case studies, E=examples, X=none • V ^d =Various/unspecified count •
S12* [22]	2024	45	✓	4L	92	ER ^j data model for unifying SC ⁱ vulnerability taxonomies
S13 [23]	2025	20+	✓	5L	X	Traces vulnerabilities in real attacks
Our Work 2025 24+ ^f ✓ Cause 50 Real-world validation, root-cause framework from incidents						

Footnotes: * SoK ^aTax: xL=x-Layer, xC=x-Category FM=Formal Methods, DT=Detection-based, RS=Root-cause-based, TB=Tool-Based, HR=Hierarchical, UN=Unified • ^bPrev: ✓=comprehensive, P=partial, X=none • ^cWild:

- Numbers=exploit count, C=case studies, E=examples, X=none • V^d=Various/unspecified count •
- NET/CON/SC/PRO/AUX-Network/Consensus/Smart Contract/Protocol/Auxiliary • ^f 24 core + 5 deprecated vulnerabilities • ^g BE=Bugs/Errors, RC=Rogue Contracts, EX=Extortion, ED=External Data, BD=Backdoor •
- ODC=Orthogonal Defect Classification • ⁱSC=Smart Contract • ^jER=Entity-Relationship

like Slither [8] and Echidna [9] or even execution frameworks to run a swarm of analysis tools on a contract, such as SmartBugs [10]. Despite their usefulness in detecting certain classes of vulnerabilities [11], both the severity (total funds stolen) and number of successful attacks have increased in recent years [5].

Related SoKs and Surveys. Table I presents a review of three Systematization of Knowledge (SoK) and ten Systematic Literature Review (SLR) papers on Decentralized Application (DApp) security. Surveys and SoKs classify tens of vulnerabilities across different layers of the blockchain stack (e.g., network, smart contract, and protocol layers) [12], [17], [18]. These taxonomies have become increasingly granular in classifying weaknesses [17] and mapping them to formal verification properties [15], or aligning them with frameworks such as NIST bugs [14]. However, our review unveils a few gaps in this literature: First, *prevention gap*: only 4 of 13 works (S4, S9, S12, and S13) address prevention methods, most of which focus exclusively on detection. Second, the *validation gap*: merely 3 works validate their findings against real-world exploits—S3 (case studies), S9 (181 exploits) and S12 (92 exploits), while others *rely on synthetic datasets*. Third, the *taxonomy evolution* shows a shift from simple categorization to sophisticated multi-layer frameworks (4L, 5L), yet *none adopt a root-cause approach based on real incidents*.

Telltale Signs from Real-World Incidents. At 08:56 UTC

*Joint first authors with equal contribution.

on 13 March 2023, the Euler Finance lending market was solvent; four minutes later, it was \$197 million in deficit [24], [25]. Nothing in its Solidity source code could be attributed to common smart contract vulnerabilities, no unchecked casts, no arithmetic overflow, or no reentrancy bug. Instead, an unguarded auxiliary function, a liquidation discount, and a flash loan combined into an unexpected profit engine to exploit the contract. Several months earlier, the Nomad bridge was emptied of \$190M because a single initializer function had been called with the wrong argument during an upgrade, resetting its trusted root to zero address and turning every message into valid [26]. Episodes like these are not rare accidents [5]. The vast majority of existing literature and automated tooling remains heavily focused on implementation-level weaknesses, the “known unknowns” [12], [17], [15]. Yet, as incidents like the \$197M Euler Finance exploit [24] show, financially devastating attacks may not merely stem from coding mistakes. In the mentioned protocol’s case, the attack succeeded as the liquidation mechanism was flawed. The code executed exactly as intended; it was the intention itself that created the vulnerability. This incident is not an isolated case. It highlights a systemic issue: while literature focuses on finding bugs in the code, a large-scale analysis of real-world attacks and comparing the exploited vulnerabilities in these incidents and their alignment with the literature is required. Thus, we propose the following research questions:

- 1) **RQ1:** Which vulnerabilities *still* affect Ethereum smart contracts as of 2025, and how can they be mitigated?
- 2) **RQ2:** What factors, other than mere coding vulnerability patterns (identified in RQ1), are evident in the real-world incidents and contribute to the success of the attacks?
- 3) **RQ3:** What categories do these vulnerabilities fall into based on their root causes?

To address RQ1, we conduct a systematic literature review (SLR) of 71 seminal papers, synthesizing their findings into a comprehensive catalog of 24 active and 5 deprecated vulnerabilities. This provides a robust, state-of-the-art baseline of academically recognized threats. To address RQ2, we perform a deep, qualitative analysis of 50 high-impact smart contract exploits that occurred between 2022 and early 2025, representing a total of \$1 093 979 779 in losses. By examining post-mortem reports, on-chain transaction data, and the attacked protocols’ source code and documentations, we move beyond bug labels (e.g., reentrancy) to uncover the steps (*exploit chains* Section IV) that enabled these attacks. This approach yielded a novel, *four-tier root-cause* framework derived directly from our empirical analysis, complementing the academic literature (Section V). We argue that real-world exploits are best understood as originating from one of these four tiers: 1) flawed economic design and protocol logic (24%), 2) protocol lifecycle and governance failures (24%), 3) external dependency vulnerabilities (24%), and 4) implementation-level weaknesses (28%).

In summary, this paper makes the following contributions:

- 1) An up-to-date, curated catalog of 24 *active* and 5 *deprecated* smart contract vulnerabilities based on a systematic

review of 71 papers. For each active vulnerability type we include detailed prevention/mitigation strategies and real-world incidents involving (in the accompanying repository¹).

- 2) A manual analysis of 50 high-profile successful incidents² (selected from a pool of 337 incidents) occurred in 2022–2025 on various protocol types (e.g., DeFi, cross-chain bridges, etc.) demonstrating that the most severe real-world exploits stem from factors beyond mere/isolated implementation mistakes. We identify *exploit chains* that combine human, operational, and economic design flaws with implementation bugs to enable attack vectors.
- 3) A novel, *four-tier root-cause* framework for smart contract exploits, grounded on our analysis of real-world exploit methods.
- 4) A case study on a sub-category of our root-cause-based framework with defense strategies against exploits targeting this sub-category.
- 5) A dataset of real-world successful attack attributes, including exploited vulnerability types (confirmed by manual expert analysis), victim and target account addresses, and attack transaction hashes.

By systematizing the knowledge of *how smart contracts fail in practice* (and attributing these failures to vulnerabilities found in our SLR and root-causes in our four-tier framework), we show that many successful attacks stem not just from code-level bugs, but from a combination of design, governance, dependency, and operational factors. The rest of the paper is organized as follows. Section II presents our rigorous SLR and incident review protocol. Section III presents the results of our SLR for RQ1. Section IV presents the results of our real-world incident analysis insights for RQ2. Section V demonstrates our multi-tier incident root-cause framework for RQ3. Finally, Section VI concludes the paper.

II. METHODOLOGY

To answer RQ1–RQ3, we analyzed both academic literature and high-quality technical smart contract incident (successful attack) reports. Sections II-A and II-B elaborate on the protocol used to identify, collect, and review these resources.

A. Academic Literature Study Protocol

To answer RQ1, we conducted a systematic literature review (SLR) to obtain a comprehensive and structured understanding of Ethereum smart contract vulnerabilities from academic literature. Following Kitchenham’s guidelines [27], our review process includes study identification (Section II-A1) and study selection (Section II-A2).

1) Study Identification: Here, we first present the process of formulating search queries for scientific databases. Then, we present the process of filtering them to collect relevant scientific literature from reputable sources (briefly in Figure 1).

Search Keywords. We used the PICOC (Population, Intervention, Comparison, Outcome, Context) framework [27] to achieve a systematic search. According to this method, we defined keywords for our search in Table II. The “Population” was defined as the technology

Identification		Screening		Eligibility	
ACM (395), IEEE (541), Scopus (74), Springer (7,113), G-Scholar (8,890), Year (EC1)	Duplicate Removal & Language Filter English (EC2)	Title & Abstract Screening Platform (EC3), Content (EC5, EC6)	Full-text Assessment Availability (EC4)	Quality-Based Selection Empirical Validation, Novel Contributions	
17,013 Initial Records	14,328 Papers	2,186 Papers	350 Papers	71 Papers	

Fig. 1: Systematic review filtering process showing the reduction from 17 013 identified papers to 71 quality-validated papers.

TABLE II: Keyword extraction method

Cat. Keyword	Synonyms
P “Ethereum smart contracts”	“Ethereum contracts”, “Solidity”
I “Vulnerabilities”	“Security Flaws” OR “Security Weaknesses”
O “Mitigation”	“Classification”, “Mitigation”, “Prevention”, “Categorization”, “classification”, “taxonomy”

(smart contracts, blockchain), and the “Intervention” as the problem being investigated.

For search keywords, we added synonyms to the main keywords for varied searches. The final query for searching in scientific databases is the result of concatenating all keywords for *P*, *I*, and *O* (and their synonyms) from Table II.

Study Sources. We queried five databases: ACM, IEEE Xplore, Scopus, Springer, and Google Scholar (for pre-prints, theses, and gray literature) to collect studies. Figure 1 presents the number of papers retrieved from each: ACM (395), IEEE (541), Scopus (74), Springer (7113), and Google Scholar (8890).

2) **Study Selection:** Following the initial identification of 17 013 papers, we removed duplicates and filtered them based on the exclusion criteria in Table III. Specifically, language (EC2) and publication year 2018–2024 (EC1) filtering reduced the count to 14 328 papers. Title and abstract screening using criteria EC3 (Ethereum focus), EC5, and EC6 (topic relevance) yielded 2186 papers. Full-text assessment based on availability (EC4) further refined the selection to 350 papers. Finally, quality-based evaluation via manual review of the paper and identification of novel contributions resulted in 71 papers for in-depth analysis. Figure 1 illustrates this process.

B. Real-World Incident Review Protocol

To answer RQ2, we reviewed post-mortem incident reports. To ensure the relevance of the successful hacks and maintain quality control over the reviewed resources, we applied a four-stage incident review protocol in Figure 2.

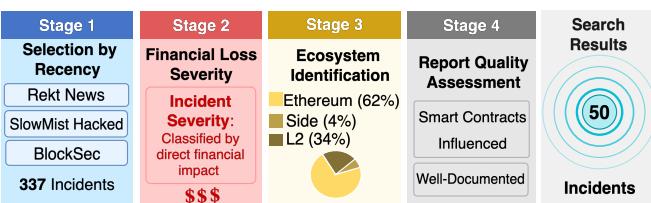


Fig. 2: Four-stage smart contract incident review protocol to systematically identify, prioritize, and analyze relevant attacks.

Stage 1: Comprehensive Data Collection. We aggregated incident reports from four reliable blockchain security sources: Rekt News [28], REKT database [29], SlowMist Hacked database [30], and BlockSec reports [31]. From each source, we extracted up to 100 of the most recent attacks based on the occurrence date. The report sources are selected for their report quality and analytical depth. In BlockSec’s case, our temporal parameters (April 2022 to April 2025) yielded only 37 documented attacks meeting our criteria, all of which were incorporated into our analysis. As a result, our initial dataset comprised a total of 337 attacks for analysis.

¹<https://github.com/HadiRe/SoK-Root-Cause-of-Smart-Contract-Incidents>

²Distinct incidents from the our real-world examples in contribution 1.

TABLE III: List of exclusion criteria

ID	Exclusion Criteria
EC1	The article was published before 2018. ^a
EC2	The article is written in a language other than English.
EC3	The article focuses on vulnerabilities in platforms other than Ethereum.
EC4	There is no full-text version of the article available.
EC5	The article is a book chapter or editorial notes.
EC6	The article does not concern any of these topics: survey of smart contract vulnerabilities, security of smart contracts, vulnerability detection or mitigation methods for smart contracts, demonstrates findings for smart contracts through simulation or empirical evidence.

^a The vulnerabilities before 2018 are included via more recent papers.

Stage 2: Impact-Based Prioritization. Incidents underwent prioritization based on financial impact severity, emphasizing attacks resulting in substantial monetary losses. This criterion ensured our analysis concentrated on vulnerabilities manifesting the most significant ecosystem consequences.

Stage 3: Ecosystem Classification. We focused on incidents occurring within the Ethereum blockchain and its compatible networks. Each case was evaluated for relevance, resulting in a final dataset of 50 distinct incidents that offer sufficient breadth while maintaining ecosystem coherence. We then mapped each incident to its respective blockchain environment, as shown in the network column of Table V, establishing three network categories:

- 1) **Ethereum Mainnet:** The Ethereum network where native consensus exist, identified by transactions on etherscan.io (31 attacks).
- 2) **Ethereum Layer 2 Solutions:** These include Arbitrum, Optimism, Base, zkSync, Linea, Scroll, StarkNet, and Mode (17 attacks). L2s inherit security from Ethereum while processing transactions off-chain [32]. They use either Optimistic or Zero-Knowledge rollups to post transaction data back to Layer 1.
- 3) **Ethereum Side-Chains:** Networks like Polygon, BSC (Binance Smart Chain), and Ronin that maintain independent consensus mechanisms while supporting Ethereum Virtual Machine (EVM) compatibility [33] (2 attacks).

This classification follows the technical distinction that true L2 solutions derive their security from Ethereum itself, while side-chains operate with separate validator sets and consensus mechanisms. For example, Base is considered an L2 because it uses the OP Stack to inherit Ethereum’s security properties, whereas Polygon PoS uses its own validator set for consensus, making it a side-chain.

Stage 4: Vulnerability Verification and Report Quality Assessment. In the final step, we applied two strict checks: (1) we made sure the incident involved at least one smart contract vulnerability not merely like infrastructure failure, human error, or stolen private keys; and (2) we checked that the incident was well-documented. To be included, a report had to clearly explain what the vulnerability was (including vulnerability localization information), how it was exploited, and what caused it. We excluded any cases that lacked enough technical detail. This process yielded a final dataset of 50 well-documented, high-impact smart contract incidents with at least one smart contract vulnerability exploited in each incident. We use this dataset as the basis for our analysis (Section IV), which later resulted in a root cause-based four-tier classification of RQ3 (Section V). We cite a total number of 126 reviewed reports for RQ2 and RQ3.

III. CURRENT ETHEREUM SMART CONTRACT VULNERABILITIES AND THEIR PREVENTION/MITIGATION

To answer RQ1, we conducted a comprehensive review of the literature (Section II-A), identifying 29 distinct smart contract vulnerabilities. Upon further analysis, we determined that 5 of these vulnerabilities have become obsolete by protocol-level upgrades (Section III-B4 and Section III-B5), compiler improvements (Section III-B2 and Section III-B3), or re-contextualized based on widespread usage (see Section III-B1). This left 24 active smart contract vulnerabilities based on the academic literature. For each of the active vulnerabilities, we collected prevention and mitigation strategies from a combination of academic sources and technical reports confirmed by expert review (Section III-A).

A. Active Vulnerabilities

Our SLR resulted in the conclusion that a core set of 24 vulnerabilities remains exploitable and economically attractive. We unpack each of these *active* threats here. For every vulnerability we (i) recap a real-world incident or a proof-of-concept vulnerability (full information is included in our companion repository), (ii) expose the coding pattern that enables it, and (iii) distill the prevention or mitigation techniques. Taken together, this section provides a current, practitioner-oriented catalog for anyone designing, reviewing, or formally verifying Ethereum smart contracts.

1) Integer Over/Underflow and Rounding Errors: These vulnerabilities stem from Solidity's limitations in handling of numerical data types. *Integer Over/Underflow* vulnerability arises when the result of a calculation exceeds the lower or upper limits of the variable type size, making it impossible to be expressed within that type. This is common with smaller data types. For instance, in Listing 1, the balance stored in balances mapping will reset to zero if it reaches the unit value of the upper limit 2^{256} . Additionally, *rounding errors* occur when division operations are performed on integers in Solidity, which leads to inconsistent results of calculations, especially when precision is required [34].

```

1 contract IntegerOverflowExample {
2     mapping(address => uint256) public balances;
3     uint256 public constant MAX_UINT = type(uint256).max;
4     function deposit(uint256 amount) public {
5         require(amount > 0, "Amount must be > zero");
6         balances[msg.sender] += amount;
7     }
8     function withdraw(uint256 amount) public {
9         require(amount>0, "Amount must be greater than zero");
10        require(balances[msg.sender]>=amount, "Insufficient");
11        balances[msg.sender] -= amount;
12    }
13    function forceOverflow() public {
14        balances[msg.sender]=MAX_UINT;
15        balances[msg.sender]+=1;
16    }

```

Listing 1: Integer overflow/underflow in Solidity

In the listing 1, the function `forceOverflow` attempts to assign the maximum possible value to a user's balance and then increase it by one, which in earlier Solidity versions (<0.8.0) would have resulted in an integer overflow. Similarly, the `withdraw` function reduces the user's balance, which in older versions, if unchecked, could lead to an integer underflow. Solidity 0.8.0 and later versions mitigate these issues by introducing built-in overflow and underflow checks, making such operations revert automatically.

Mitigation/Prevention. To prevent arithmetic vulnerabilities in Solidity smart contracts, developers commonly rely on SafeMath [35], built-in checks in Solidity 0.8.0+, and compiler warnings. When an overflow or underflow occurs, the SafeMath library reverts the transaction. However, developers may apply it inconsistently, and it increases gas costs. Solidity 0.8.0+ eliminates the need for SafeMath by performing these checks at the compiler level, though it still introduces some runtime overhead [36], [34]. To prevent precision loss, developers use FullMath library [37]. Another prevention strategy in case of integer over/underflow is using suitable data types [36].

Using formats with sufficient capacity, such as `uint256` instead of `uint8` helps prevent integer over/underflow issues.

2) Improper Handling of External Calls: Also known as “Mishandled Exception”, “Exception Disorder”, or “Unchecked Call Return Value”, this vulnerability arises when a contract ignores the boolean success flag returned by a low-level external call. In the EVM the operations `call`, `delegatecall`, `staticcall`, `callcode`, and the wrappers `send` and `transfer` never propagate a revert from the callee; instead they return a tuple (`success`, `data`), where `success` is true on completion and false if the callee runs out of gas, deliberately `reverts`, or encounters any other error. Failure to test this flag allows the caller to proceed with an invalid state, potentially locking Ether or corrupting storage.

Two particularly common manifestations are documented in the literature [38], [39]. First, when Ether is transferred with `send` or `transfer`, only a 2300-gas stipend is forwarded. Any recipient whose fallback or receive function requires more gas will revert, but the sender sees only a false return value. If the caller ignores that flag, the Ether remains trapped in the sending contract, sometimes irreversibly. Second, any low-level call—whether it uses `call`, `delegatecall`, `staticcall`, or `callcode`—that proceeds without checking success suffers the same silent-failure risk: the callee's error is suppressed, control returns to the caller, and subsequent logic executes under false assumptions.

Prevention/Mitigation. To mitigate vulnerabilities related to external calls, developers should verify the return value of such calls. For example, using `require(sendResult, "Transfer failed")` after `send`. Additionally, it is recommended to use `transfer` instead of `send`, as `transfer` automatically reverts the entire transaction in case of failure [38]. Since Solidity 0.6.0, the introduction of the `try/catch` structure has provided a more robust way to handle errors in external calls. Furthermore, adopting well-established standards such as ERC20 and ERC721, which implement secure transaction mechanisms, is advised [40].

3) Reentrancy: The reentrancy vulnerability occurs when functions are maliciously reentered (e.g., through fallback functions) [41]. If the attacker is able to bypass the validity checks in the target contract, they can reenter the callee function maliciously. The vulnerability arises due to two main reasons: first, a contract's control flow decision relies on some of its state variables that should be updated by the contract itself, but are not, before calling another contract; and second, there is no gas limit when handing over the control flow to another contract. The Decentralized Autonomous Organization (DAO) attack exploited this vulnerability in 2016. As shown in Figure 3, the original contract includes functions for depositing and withdrawing Ether. The primary issue lies in the `withdraw` function, where the user's balance is set to zero after Ether has been sent to their address. This sequence allows an attacker to repeatedly call `withdraw` before the balance update occurs for multiple unauthorized withdrawals. In the attack contract, the attacker leverages the `fallback` function to invoke `withdraw` repeatedly before the balance is adjusted, receiving multiple illegal Ether payments.

Mitigation/Prevention. Two well-established techniques, Check-Effects-Interactions (CEI) [42], [43] and reentrancy guard locks, are commonly used to mitigate this issue. In the vulnerable `DepositFunds` contract (Listing 2), in line 10, the balance is modified after the external call returns, enabling the malicious reentrancy. In CEI, the contract first verifies conditions (Check), then executes external interactions (Interactions), and finally updates the internal state (Effects). In the improved contract, `SecureDepositFunds` (Listing 4), which implements the CEI pattern, the user's balance is reset to zero before transferring the Ether. This ensures that the internal state is updated first (Effects), followed by external interactions (Interactions). By verifying the conditions initially (Check) and ensuring that no funds remain in the account before initiating the transaction, the contract prevents an attacker from making repeated withdrawals even

```

1 contract DepositFunds {
2   mapping(address => uint) public balances;
3   function deposit() public payable {
4     balances[msg.sender] += msg.value;
5   }
6   function withdraw() public {
7     uint bal = balances[msg.sender];
8     require(bal > 0);
9     (bool sent,) = msg.sender.call{value: bal}("");
10    balances[msg.sender] = 0;
}

```

Listing 2: Vulnerable Contract

```

1 contract Attack {
2   DepositFunds public depositFunds;
3   constructor(address addr) {
4     depositFunds = DepositFunds(addr);
5   }
6   fallback() external payable {
7     if (address(depositFunds).balance >= 1 ether) {
8       depositFunds.withdraw();
9     }
10    function attack() external payable {
11      depositFunds.deposit{value: 1 ether}();
12      depositFunds.withdraw();
}

```

Listing 3: Attack Contract

```

1 contract SecureDepositFunds {
2   mapping(address => uint) public balances;
3   function deposit() public payable {
4     balances[msg.sender] += msg.value;
5   }
6   function withdraw() public {
7     uint bal = balances[msg.sender];
8     require(bal > 0); balances[msg.sender] = 0;
9     (bool sent,) = msg.sender.call{value: bal}("");
10    require(sent, "Failed to send!");
}

```

Listing 4: Fixed Contract

Fig. 3: Vulnerable \leftrightarrow attack contract pairs and the patched contract involving in reentrancy vulnerability.

if they attempt to exploit reentrancy [44].

The second approach involves using a lock variable to prevent multiple executions of a sensitive function at the same time. When withdraw is called, the lock is activated to prevent reentry into the function until execution is complete. Once the function finishes, the lock is released, ensuring that no concurrent executions occur. This mechanism effectively neutralizes reentrancy attacks by blocking repeated calls to the function within the same transaction cycle.

4) Dangerous Delegatecall: This vulnerability gained attention after the Parity hack in 2017, which exploited the EVM opcode `delegatecall`. Unlike a regular `call`, `delegatecall` executes the callee's bytecode while retaining the caller's storage, `msg.sender`, and `msg.value` (Figure 4). Consequently, if the address provided to `delegatecall` points to untrusted code (code section of contract B in Figure 4), that code runs with full write access to the caller's state and can arbitrarily modify storage or even trigger self-destruction, leading to loss of funds or contract functionality [45], [12].

A critical flaw was disclosed in December 2023 after several projects adopted the meta-transaction standard ERC-2771 [46]. Many implementations used `delegatecall` to let the forwarder library determine `_msgSender()`, but failed to validate the forwarded payload. An attacker could therefore wrap arbitrary calldata inside a trusted forwarded request and trick the callee into believing that the transaction originated from any address of the attacker's choosing. The technique was observed in the wild, causing losses of 84.59 ETH and 17 394 USDC in separate incidents [47], [48].

Mitigation and Prevention. The safest way to reuse code is to rely on *stateless* Solidity libraries that are linked at compile time with the `using for` directive; the bytecode of such a library executes in its own context, so any `SSTORE` or `SLOAD` it performs affects only the caller's storage slot that the in-line assembly refers to, not the library itself [49]. When dynamic dispatch is unavoidable, restrict `delegatecall` to logic contracts that you have fully audited and whose bytecode hash is hard-coded or stored behind an owner-controlled upgrade mechanism. In practice, this means keeping the implementation address in an immutable variable or

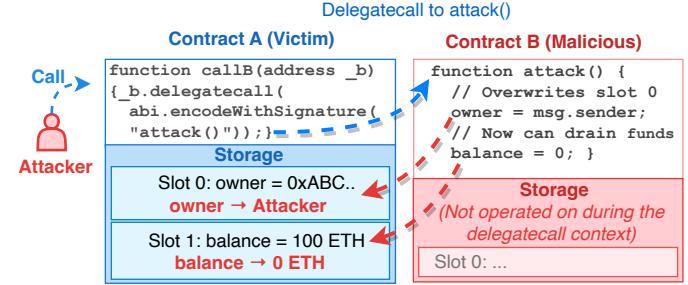


Fig. 4: Delegatecall attack flow. A flawed interaction through a `delegatecall` can be initiated by the attacker calling a function in the target contract, which involves an unsafe `delegatecall` to an attacker-desired contract. This vulnerability might require a chain of exploits to pull off a successful attack (see Insight 3).

in a proxy's implementation slot that can be changed only by a multisig with `onlyOwner` or `AccessControl` [50]. Never allow arbitrary user-supplied addresses to be the target of `delegatecall`. Upgradable contracts add a second risk surface: if the new implementation introduces or reorders state variables, every storage slot after the change shifts, corrupting data. To avoid this, adhere to an explicit storage-layout discipline: reserve *gaps* (`uint256[50] private __gap;`) at the end of each contract, append new variables only after those gaps, and use advanced Solidity code differencing tools such as SOLIDIFFY [51] before every upgrade to verify the intention of the modified codes. Standards that encode these patterns, notably EIP-2535 (Diamond) and the UUPS proxy, embed storage-slot constants and upgrade guards that make layout violations detectable on-chain [52], [53]. Leveraging audited reference implementations such as the OpenZeppelin diamond and UUPS modules eliminates common flaws around `delegatecall` and proxy upgrades [54] (that use `delegatecall` to upgrade the contract logic).

5) Unbounded Loops in Dynamic Arrays: When a loop in a smart contract runs over all elements of a dynamic array, the gas cost for executing the loop will be directly related to the number of elements. If the array grows over time or contains many elements, the gas cost may exceed the block gas limit. This can lead to the failure of the function's execution and result in a DoS attack.

Mitigation/Prevention. Developers should follow best practices like limiting loop iterations [55], dividing operations into smaller transactions, employing pull payment patterns rather than direct asset transfers in loops, and tracking gas consumption (before deployment) to improve efficiency [43], [56], [57].

6) DoS Attack via Owner Account: Many smart contracts have an owner account that controls the contract. If this account is not adequately protected, attackers may exploit it, which could lead to severe consequences, such as permanently locking Ether in the contract [58].

Mitigation/Prevention. Contracts should restrict access to critical functions multi-signature approvals for transactions to avoid reliance on a single account for key operations [59].

7) State-Revert: A contract is vulnerable to *state-revert abuse* when it reveals the result of a state-changing operation *before* the change becomes irrevocable, allowing a caller to decide whether to let the transaction stand. The classic setting is a probabilistic reward dApp that pays one of several prizes: an attacker routes the call through an auxiliary contract, records its own balance, invokes the reward function with `(bool ok, bytes data) = address(dApp).call(payload)`, inspects `data` to see which prize was drawn, and immediately executes `require(false)` if the prize is unsatisfactory. Because the revert propagates upward, every write inside the dApp, including token transfers, is rolled back, yet the attacker has already learned the outcome. By repeating this cherry-picking loop until the desirable branch appears, the adversary collects only high-value rewards and skews the game's expected

value [60].

Prevention/Mitigation. Effective countermeasures postpone disclosure or make it irreversible. Commit-and-reveal schemes write a sealed commitment (`keccak256(seed, user)`) in one transaction and reveal the seed in a later block, so the outcome is fixed before either party knows it. When genuine randomness is needed, verifiable random-function services (e.g., Chainlink VRF) provide entropy that cannot be observed in time to trigger a revert. Finally, the reward function can burn gas or charge a non-refundable fee *before* emitting any tell-tale data to eliminate the economic incentive to revert [43].

8) **Frozen Ether:** *Frozen Ether* (a.k.a. “locked money”) occurs when a contract can accept ETH but, because its withdrawal logic lives in an external library reached via `delegatecall`, it cannot release the funds once that library is absent. If the library address is wiped with `selfdestruct`—intentionally or by accident—the forwarder still receives deposits (no code is needed for that), yet every spend attempt reverts, leaving the balance permanently inaccessible [61], [45].

Mitigation/Prevention. Any contract designed to hold ETH should therefore embed an internal withdrawal function, e.g. using `call{value: amount}("")` or hard-code an immutable library address that cannot be destroyed; without such a path, deposits may become irreversible [62].

9) **Insecure Randomness:** This vulnerability affects contracts that require randomness (e.g., gambling and lottery contracts). These contracts often generate pseudo-random numbers using seeds such as `block.number`, `block.timestamp`, `block.difficulty`, or `blockhash`. Since these seeds can be influenced by blockchain network participants (validators or miners), malicious actors can manipulate these values to alter the outcome in their favor [63], [64].

```
1 contract VulnerableLottery {
2     uint256 private seed;
3     function play() public payable {
4         require(msg.value == 1 ether);
5         uint256 random =
6             uint256(keccak256(abi.encodePacked(block.timestamp,
7                 seed)));
8         if (random % 10 == 0){msg.sender.transfer(9 ether);}
7         seed = random;
}
```

Listing 5: Lottery Contract with insecure randomness.

For example, the `VulnerableLottery` [65] contract’s dependence on `block.timestamp` (Listing 5) and a private seed for random number generation results in unsafe randomness. Given that miners can alter `block.timestamp` within a narrow range, an attacker can affect the result by sending transactions at particular times. Moreover, using a predictable seed does not generate enough entropy to allow for the prediction of future outcomes.

Prevention/Mitigation. For secure random number generation, using blockchain-internal sources like `block.timestamp` and `block.difficulty` is insecure due to their transparency and predictability [66]. Instead, developers can utilize external oracles such as Chainlink VRF (Verifiable Random Function), which provides more tamper-resistant randomness [67], [68]. Another approach is the commit-reveal scheme, which enhances randomness security by requiring participants to submit a concealed value first and reveal it later, preventing attackers from predicting or manipulating results [69], [43]. Additionally, employing public-key cryptosystems like the Signidice algorithm [70], originally developed for EOS blockchain and adopted in projects like DAO.Casino, can generate random numbers in two-party contracts. Furthermore, cross-chain oracles such as BTCTRelay [71] provide a trustless mechanism for Ethereum smart contracts to verify Bitcoin transactions. In order to enable the use of Bitcoin’s Proof-of-Work-derived entropy as a source of randomness in Ethereum contracts, BTCTRelay stores Bitcoin block headers on Ethereum and permits contracts to check Bitcoin transaction inclusion proofs. When combined with other

protections against validator manipulation, such as time-delay and multi-source entropy, this strategy is useful.

10) **Authorization Using tx.origin:** The global `tx.origin` holds the externally owned account (EOA) that initiated the entire transaction chain, whereas `msg.sender` refers to the immediate caller of a function. Using `tx.origin` in access control checks rather than `msg.sender` lets an attacker insert a malicious contract into the call flow and pass the check as the original EOA.

In Listing 6, the `Victim.withdrawAll` function authorizes withdrawal with `require(tx.origin == owner)`. The attacker contract in Listing 7 simply calls `withdrawAll`; because `tx.origin` still equals the owner’s EOA, the transfer succeeds and forwards funds to the attacker. No owner approval occurs.

```
1 contract Victim {
2     address public owner;
3     constructor() {owner = msg.sender;}
4     function withdrawAll() public {
5         require(tx.origin == owner, "Not authorized");
6         payable(msg.sender).transfer(address(this).balance);
7     }
8 receive() external payable {} }
```

Listing 6: Victim contract demonstrating the use of `tx.origin`

```
1 contract Attacker {
2     Victim public victimContract;
3     constructor(address _victimContractAddress) {
4         victimContract = Victim(_victimContractAddress);
5     }
6     function attack() public { victimContract.withdrawAll();
7 } receive() external payable {} }
```

Listing 7: Attacker contract exploiting `tx.origin` vulnerability

Prevention/Mitigation. Developers should avoid using `tx.origin` for authorization and instead rely on `msg.sender`, which represents the immediate caller of the function [72]. Furthermore, it can be beneficial to restrict function access to approved users [73] by putting strong access control mechanisms such as OpenZeppelin Ownable contract.

11) **Front-Running (Abusing Transaction-Ordering Dependency):** Front-running is a common attack vector that exploits the public and sequential nature of blockchain transaction processing, allowing an adversary to profit by preempting a victim’s pending operation. Because miners (or validators) decide the execution order of pending transactions, an adversary who foresees a profitable state change can insert their own call ahead of the victim’s. This is done either by rebroadcasting the same call with a higher gas price, which miners typically favor [12], or if the adversary controls the block proposer by unilaterally reordering transactions regardless of fees [74]. Such re-ordering can alter contract outcomes and extract value from honest users. In a simple scenario of front-running attack, Alice broadcasts transaction T_1 with gas price G_1 at time t_1 . Bob observes T_1 in the mempool and immediately submits T_2 with a higher gas price $G_2 > G_1$ at $t_2 > t_1$. Validators, who generally sort pending transactions by gas price, include T_2 first in block B_x ; T_1 is confirmed later (in the same or a subsequent block). By pre-empting the state change encoded in T_1 (e.g., a profitable DEX swap), Bob captures its economic value, an archetypal front-running attack that exploits transaction-ordering dependency.

Prevention/Mitigation. To address this vulnerability, developers can use techniques like batch auctions, which process several transactions at once to limit manipulation of transaction ordering [75], and commit-reveal schemes, which divide transactions into two phases to initially conceal sensitive information and reduce front-running risk [75].

12) **Prodigal Contract:** This vulnerability refers to smart contracts that wrongly send Ether to arbitrary addresses and insufficiently control Ether transfers. It is also known in literature as “leaking ether to arbitrary addresses”. During runtime, sending Ether to addresses other than contract’s owners or addresses that have not deposited

Ether to the contract, could flag the existence of this vulnerability in a contract [76].

Prevention/Mitigation. Thorough input validation and stringent access rules are necessary to stop prodigal contracts. Unintentional asset leaks are decreased via input validation, which makes sure parameters like recipient addresses or transfer amounts follow certain rules (such as whitelisting addresses or confirming non-zero values) [36]. Another security measure against unauthorized activities is added by restricting important functions (e.g. fund transfers) to authorized entities through the use of Role-Based Access Control (RBAC) or OpenZeppelin’s `onlyOwner` modifier [54].

13) Ether Lost to Orphan Address: To initiate an Ether transfer, a valid 160-bit address must be specified. If the provided address is invalid or does not exist, the transferred funds will be permanently lost [39]. If the `to` address is set to the contract’s own address, the transferred tokens will get locked inside the contract with no way to withdraw them, resulting in a loss. This happens because the function does not verify whether the `to` address is a valid recipient, including ensuring that it’s not the contract itself (see Listing 8).

```
1 function transfer(address to, uint256 amount) public
2     returns (bool) {
3     balances[msg.sender] -= amount; balances[to] += amount;
4     emit Transfer(msg.sender, to, amount); return true;
4}
```

Listing 8: Transfer function without destination address validation

Prevention/Mitigation. Developers should assure (via `require` statements) recipient addresses are neither the zero address (`address(0)`) nor the contract’s own address (`address(this)`) and confirm the recipient’s capacity to manage transfers (e.g., avoiding non-withdrawable contracts) [54]. Adopting “pull-over-push” withdrawal mechanisms, where users initiate transfers rather than relying on direct pushes, also helps reducing risks of sending to invalid addresses [77].

14) Untrustworthy or Manipulable Data Feeds: Smart contracts often depend on external facts such as token prices, weather data, and sports scores to make irreversible decisions. An *oracle* is the component that transports those facts on chain (or acts as a service contract to provide data from other on-chain services). Recent studies show that integrity of the data can be manipulated at stages of its lifecycle [4]. We identified two settings where data feeds of these oracles are manipulable:

- *Decentralized Exchange (DEX)-based on-chain oracles.* *System model:* a contract queries the current price shown by an Automated Market Maker (AMM) pool such as Uniswap [78]. *Attacker model:* participants may inject capital (e.g., via a flash loan) to shift the pool’s reserves for a single block and read back a distorted price before arbitrage restores equilibrium [79].
- *Off-chain oracles.* *System model:* trusted reporters sign values off-chain and the oracle publishes them on-chain. *Attacker model:* compromise, bribe, or simply delay reporters to feed stale or false data [80].

Prevention/Mitigation. The literature proposes the following defense strategies:

- *Temporal aggregation.* To deter price manipulation in DEX-based oracles, use multi-block medians or long-window time-weighted average pricing (TWAP) taken to smooth out single-block distortions [4].
- *Economic deterrence.* In case of off-chain oracles, require the reporters to put down a bond larger than the profit they could earn by lying, and slash the bond on proven fraud [80], [4].
- *Grace periods for finality.* A contract should ignore data that has not yet cleared a *dispute* window to block contested updates.
- *On-chain invariants and circuit breakers.* Use invariants (in form of `require` statements) that cap per-block price changes to stop attacks at runtime [81].

- *Real-time monitoring.* To detect price manipulation attacks, dynamic monitoring systems such as rule-based (DeFiRanger [79]), behaviour-model (DeFort [82]), and LLM-based (DeFiScope [83]) detectors raise alerts, while counter-measure agents like FlashGuard [84] can front-run malicious transactions.

Based on the mentioned two system settings, combining smoothing, economic incentives, program invariants, and runtime monitoring can raise the adversary’s required capital, coordination, and technical difficulty beyond profitable levels.

15) Compiler Version Not Fixed: This vulnerability occurs when a contract uses an outdated or broadly specified compiler version [85]. For example, specifying `pragma solidity ^0.4.0;` allows compilation with any version from `0.4.0` to `0.5.0` (excluding `0.5.0`). This makes it possible for the contract to be compiled with versions known for certain vulnerabilities. Here, in versions before `0.4.22`, if a function intended as a constructor did not match the contract name exactly (e.g., after renaming the contract but not the function), it was treated as a regular public function, not a constructor. This allowed attackers to invoke it maliciously after deployment. By fixing the compiler version explicitly (e.g., `pragma solidity 0.4.25;`), the contract becomes compilable only with `0.4.25`, where the `constructor` keyword is mandatory, thus mitigating this vulnerability [12].

Prevention/Mitigation. Developers can use code linters [86] to find missing version specifications that explicitly state the compiler version (e.g., `pragma solidity 0.8.17;`). Furthermore, developers should review Solidity release notes prior to upgrading the project to prevent breaking changes [36].

16) Event-Ordering Bug: If distinct sequences of transactions or function invocations lead to an unexpected behavior or final state (post-transactions), it is identified as an EO bug [87].

Prevention/Mitigation. Developers should impose strict execution order by pre- and post-conditions (`require/assert` statements in entry and exit points of functions in Solidity) to ensure transactions follow a desired sequence to prevent event-ordering (EO) issues [43], [87].

17) Type Casting: This vulnerability stems from improper type conversions in Solidity, such as truncation bugs when casting a larger integer type (e.g., `uint16`) to a smaller one (e.g., `uint8`), leading to data loss. Additionally, converting between signed and unsigned types of the same width can cause “signedness bugs,” where negative values turn into large positive ones or vice versa [88], [34].

Prevention/Mitigation. To prevent type-casting vulnerabilities, developers should ensure proper type validation by explicitly checking types and validating value ranges before performing conversions. This reduces the risk of unintended truncation [89]. Additionally, Solidity requires explicit casting to help reduce the risk of implicit conversions, especially between signed and unsigned numbers, which might introduce unpredictable behavior [90]. Refactoring the project to the most recent Solidity version is also beneficial as newer versions support type safety features [91].

18) Ponzi Scheme: In Ponzi contracts, vulnerabilities arise (for the users of the Ponzi services) due to low transparency and deceptive promises of high returns, and legal noncompliance. These schemes commonly rely on a constant influx of new investors to create an illusion of profitability, with returns paid to earlier participants using the funds contributed by newer ones, rather than from actual profits. This scheme is unsustainable and leads to losses for later investors when the inflow of new investments slows or stops [92].

Prevention/Mitigation. User education is important and users must avoid them by spotting characteristics of Ponzi schemes, such as unsustainable returns and a lack of transparency (unavailable, unverified, or obscured contract source code) [93].

19) Storage Collision: Storage Collision occurs when two contracts accessing the same storage space have a different understanding of the storage layout [94], [95], [96], [97]. This vulnerability is often

observed when a proxy upgradability design pattern is used [98]. The proxy upgradability design pattern separates a contract's state from its logic, allowing the system's functionality to evolve without losing stored data [43]. In this design, a proxy contract holds the persistent state and delegates calls to an external implementation contract using the `delegatecall` opcode. Initially, the proxy directs calls to a basic implementation. Later, when new features or patches are required, a new implementation contract is deployed. By updating the proxy's reference to this new contract, the upgraded logic is activated while the original state (the users' balances, etc.) remains intact [98], [43]. A specific subcategory of this vulnerability is also known as *type confusion*. This specific subcategory occurs when the runtime context of the logic contract (e.g., in an upgradable contract) confuses the type of an object in storage with another type, leading to its incorrect utilization. For example, when a variable is allocated with one type and later accessed with an incompatible type [94], [99].

For example, the smart contract setup in Listing 9 is vulnerable to storage collision due to the improper alignment of storage slots between the Proxy and Logic contracts when using `delegatecall`. Specifically, both contracts store critical state variables in slot `0x0`, but with different interpretations: the Proxy contract uses it for visits, while the Logic contract stores initialized and admin in the same slot. When `initialize()` is called via `delegatecall`, it writes `true` (1) to initialized and sets admin to `msg.sender`, unknowingly overwriting the `visits` variable in the Proxy contract. An attacker can exploit this by re-invoking `initialize()`, since the overwritten storage allows it to be called again, setting their own address as admin. Once this is done, the attacker can execute `withdraw()` to drain the contract's funds. See Listing 9 for more details.

```

1 contract Proxy {
2     uint public visits; // Storage slot [0x0]
3     address public LOGIC; // Storage slot [ERC-1967]
4     constructor(address logicAddress) {
5         LOGIC = logicAddress;
6         (bool success, ) = LOGIC.delegatecall(
7             abi.encodeWithSignature("initialize()"));
8         require(success, "Initialization failed");
9     }
10    fallback() external payable {
11        (bool success, ) = LOGIC.delegatecall(msg.data);
12        require(success, "Delegatecall failed");
13    }
14    contract Logic {
15        bool public initialized; // Storage slot [0x0]
16        address public admin; // Storage slot [0x0] (Collision
17        issue)
18        uint[] public artworkIDs; // Storage slot [0x1]
19        mapping(address => uint) public artworkHolders; //
20        Storage slot [0x2]
21        function initialize() external {
22            require(!initialized, "Already initialized");
23            initialized = true;
24            admin = msg.sender; // Overwrites Proxy's storage slot
25            [0x0];
26        }
27        function withdraw() external {
28            require(msg.sender == admin, "Not admin");
29            payable(admin).transfer(address(this).balance);
30        }
31    }
32 }
```

Listing 9: Wallet Contract with storage collision vulnerability

Prevention/Mitigation. To avoid storage collisions in proxy patterns, keep the implementation's storage layout consistent with the proxy's reserved slots (or assign fixed slots with inline assembly, as in EIP-1967) [100]. Supplement this with exhaustive tests and fuzzers like Harvey to catch any remaining overlaps pre-deployment [96].

20) **Business Logic Flaws:** In smart contracts, business logic flaws (aka accounting errors) occur when financial calculations are inconsistent, inaccurate, or disorganized because of type errors, misinterpreted or mismanaged units (token units), or the implementation is inconsistent with expected behavior by the protocol designer [101].

One such vulnerability happened in Vader protocol [102], where a token unit mismatch occurs when token values are incorrectly combined without the appropriate unit conversion, resulting in an error in the liquidity calculation (`calcLiquidityUnits` in Listing 10). In this case, the scales of the base and token tokens differ.

Nevertheless, they are added together in the final formula without the proper conversion, which causes users to lose money and receive an inaccurate liquidity (Listing 10).

```

1 contract Pools {
2     function addLiquidity(address base, address token,
3                           address member)
4     external returns (uint liquidity) {
5         uint addedBase = getAddedAmount(base, ...);
6         uint addedToken = getAddedAmount(token, ...);
7         liquidity = calcLiquidityUnits(addedBase, totalBase,
8                                         addedToken, totalToken, totalLiquidity);
9         liquidity [...] [member] += liquidity;
10        totalBase += addedBase;
11    }
12    function calcLiquidityUnits(uint b, uint B, uint t, uint
13                               T, uint P)
14    external view returns (uint) {
15        uint part1 = (t * B);
16        uint part2 = (T * b);
17        uint part3 = (T * B) * 2;
18        uint _units = (((P * part1) + part2) / part3);
19        return (_units / one; )
20    }
21 }
```

Listing 10: Smart Contract: Pools

Prevention/Mitigation. Developers should write extensive test cases covering important business logic scenarios to validate code behavior. This practice helps in identifying discrepancies between the intended and actual functionalities of the smart contract. Extensive testing using model-based testing can deter business logic vulnerabilities [103], [104]. Implementing strict input validation ensures that only correctly formatted and expected data is processed by the contract to prevent unintended behaviors arising from malicious inputs [64].

21) **Dangerous Balance Inequality:** Input validation and invariant preservation are checked/enforced using `assert` and `require` statements in Solidity. Solidity smart contracts on Ethereum alone include more than 4.5 of these unique checks [105]. In such condition checks, using “`=`” instead of “`≥`” (in statements such as $a \geq b$) when verifying inventory arises from the flawed assumption that the exact match will always occur. This practice can lead to a vulnerability, as even minor, unexpected changes in inventory values, whether intentional or accidental, could bypass verification logic [85].

Prevention/Mitigation. Developers should use “`≥`” to account for potential pre-existing balances [106]. Slither [8], the static analysis tool, provides a “dangerous-strict-equality” detector to identify this vulnerability. Moreover, Continuous Integration (CI) pipelines can automatically check the semantic strength of a pre-/post-condition check in Solidity using SINDI [107].

22) **Resource Exhaustion:** The gas mechanism in EVM is used to determine how much it costs to execute smart contracts. This system should guarantee that each instruction's execution cost is calculated according to the resources it uses, including CPU, RAM, and I/O. Nevertheless, some processes, such as `blockhash` and `balance`, which require access to blockchain-stored data, are substantially underpriced compared to their actual computational and storage impact due to inconsistent gas pricing for specific EVM instructions [109]. One practical example is looping through a list of accounts to repeatedly retrieve their `balance`. Since the gas cost for this operation is lower than its actual computational overhead, an attacker can craft transactions that continuously invoke this instruction, causing a significant workload on the network with *minimal gas expenditure for the attacker*. This shows how inconsistencies in EVM's resource metering (vs opcode gas) can be leveraged to create transactions that disproportionately consume computational power.

Prevention/Mitigation. Ethereum clients have undergone several upgrades to mitigate resource exhaustion risks. In May 2024, a flaw was exposed in the `Geth` client's `FeeHistory` interface that allowed RPC requests with up to 600k `rewardPercentiles` due to missing limits, leading to excessive memory usage and node crashes. The issue was resolved by capping the number of `rewardPercentiles` [110], [111].

23) **Access Control:** An access control vulnerability occurs when a smart contract improperly manages access to critical function-

TABLE IV: Summary of deprecated smart contract vulnerabilities and their mitigation

Vulnerability	Sec.	Deprecation Reason(s)
Upgradable Contracts	§III-B1	Re-contextualized from a vulnerability to a technical feature. Its risks are manageable through robust on-chain governance and verifiable evolution patterns.
Wrong Address	§III-B2	Mitigated by ABI-decoding requirements in Solidity v0.5.0. Calls with <code>calldata</code> shorter than the expected arguments will revert, preventing the padding that enabled the attack.
Erroneous Visibility	§III-B3	Addressed by Solidity v0.5.0, which mandated explicit function visibility declarations (<code>public</code> , <code>private</code> , etc.).
Suicidal Contract	§III-B4	The behavior of the <code>selfdestruct</code> opcode was severely curtailed by EIP-6780. It now only functions if executed in the same transaction as contract creation, neutralizing its use in common attack patterns.
Call-stack Depth Limit	§III-B5	Rendered economically infeasible by the Tangerine Whistle hard fork (EIP-150) [108], which introduced the “63/64 gas rule.” This rule limits the gas forwarded in a call, preventing deep recursion attacks.

alities, allowing unauthorized users to carry out operations including asset transfers, ownership changes, and significant contract settings modifications. Usually, this vulnerability results from the lack of a suitable authentication method. Usual ways to authenticate include implementing Role-Based Access Control (RBAC) [112] using `require` statements at the entry point of the function body or using a Solidity `modifier`. This vulnerability is also frequently caused by defining functions as `public` when only the contract owner or authorized users should be able to access them, failing to implement access control conditions like `require(msg.sender == admin)` and exposing sensitive variables like `owner` or `balance` as `public` when they should be `private` to prevent unwanted access [113], [114], [115]. For example, because there is no `require(msg.sender == owner, "Not authorized")` check in Listing 11 (after line 5), any user can alter the contract owner using the public function `changeOwner`. Likewise, the `withdraw` function has no access restriction, so anyone can take money out without being authenticated.

```
1 contract AccessControlVulnerableContract {
2   address public owner; uint256 public funds;
3   constructor() { owner = msg.sender; }
4   function changeOwner(address _newOwner) public { owner =
      _newOwner; }
5   function withdraw(uint256 _amount) public {
6     require(_amount <= funds, "Insufficient funds");
7     funds -= _amount; payable(msg.sender).transfer(_amount); }
```

Listing 11: VulnerableContract Example

Prevention/Mitigation. Developers can utilize OpenZeppelin’s Ownable and AccessControl contracts, which offer role management, to avoid access control vulnerabilities. The `onlyOwner` modifier [116] in the Ownable contract is used to limit sensitive operations to the owner role. AccessControl in this library enables defining roles and their permissions [117].

24) **Timestamp Dependence:** This vulnerability occurs when the contract’s logic depends on the value of `block.timestamp`. Due to the clock drift of distributed systems, miners are allowed to adjust the timestamp within a specific range, typically ± 900 seconds in Ethereum [118]. Smart contract condition checks may be impacted by this manipulation, even if they have nothing to do with producing random numbers. For instance, consensus participants can alter the end time of events like bids or auctions [119]. Validators can lengthen waiting or lock-up periods before specific activities to change the outcome to the advantage of a particular group [43]. Additionally, it is a typical manipulation for miners to reorganize and modify the timing of transactions, particularly in contracts that depend on the precise timing or order of transactions. Changing the timestamp may also alter the outcome of computations that depend on time, causing the contract to behave inconsistently. Miners may modify the voting schedule and outcomes to benefit themselves or a certain group, which could have an effect on governance ecosystems [120].

Prevention/Mitigation. Developers should use trusted time oracles to ensure accurate data sources [121], [122].

B. Deprecated Vulnerabilities

This section discusses vulnerabilities in smart contracts that were formerly seen as security issues but have been fixed by protocol enhancements and governance systems. Table IV presents a list of these deprecated Ethereum smart contract vulnerabilities and their brief deprecation reasons.

1) **Upgradable Contracts:** Smart contracts often incorporate upgradability mechanisms through logic contracts and proxies to manage immutability constraints of the code section of the contract [221]. Although early analyses classified upgradability as a potential vulnerability [12], recent advancements in upgradability patterns and governance mechanisms have demonstrated that this is better understood as a technical challenge rather than a security weakness. On-chain governance proposals [222] now enable decentralized control over upgrades, reducing the risk of unilateral malicious actions. Moreover, research efforts such as SoliNomic [223] have introduced formal refinement-based approaches to structuring the evolution of decentralized organizations (for correct and consistent upgrades). Therefore, upgradability, when properly designed, enhances the adaptability of DApps without compromising security.

2) **Wrong Address:** This vulnerability arises due to the improper handling of function arguments. EVM expects input arguments to be encoded in chunks of 32 bytes; however, if an attacker deliberately sends a truncated address (i.e., an address with missing bytes at the end), EVM automatically pads the missing bytes with zeros, potentially altering the intended values of other arguments. This misalignment can cause an attacker to manipulate transaction amounts or other critical parameters in smart contracts [61], [12]. The attacker could craft a transaction with a shortened address, which, when processed, causes the EVM to pad the missing bytes with zeros, leading to a shift in memory alignment where other function arguments, such as transfer amounts, are misinterpreted (e.g., excessive transfers to the attacker’s address). To prevent this vulnerability, developers should implement strict validation to check that the input data matches the expected size. In other words, the contract should explicitly check `msg.data` length to ensure proper alignment of the arguments. In modern Solidity versions ($\geq 0.5.0$), with its standard ABI decoding, the short address attack is prevented by design [224]. In these modern versions, if an attacker tries to supply a truncated address (resulting in insufficient `calldata`), the function call will revert before any parameter decoding or auto-padding occurs.

3) **Erroneous Visibility:** Erroneous visibility vulnerability emerged from Solidity’s default function visibility behavior prior to version 0.5.0. Functions without explicit visibility modifiers automatically became `public`, exposing critical operations to unauthorized external calls [225]. Implicit behavior often led to unintentional access to state-modifying, fund management, or administrative functionality. Missing visibility specifiers allowed attackers to execute privileged functions in the Parity multisig wallet incident [226].

By requiring explicit visibility declarations for all functions, Solidity version 0.5.0 released in November 2018, fixed this vulnerability [227]. Contracts without visibility specifiers are now rejected

TABLE V: Overview of studied security incidents including attack dates, financial losses in USD, and identified vulnerabilities.

ID	Incident	Date	Loss (USD)	Ecosystem	Application	Vulnerability	Network
1	Euler Fin. [123], [124], [25]	2023-03-13	197 000 000	DeFi	Lending	Access control, Business Logic Flaws	Mainnet
2	Nomad Bridge [26], [125]	2022-08-02	190 000 000	Bridge	Cross-chain	Access Control	Mainnet
3	BonqDAO [126]	2023-02-03	120 000 000	DeFi	Stablecoin	Price Manipulation	Sidechains
4	WooFi [127], [128]	2024-03-06	85 000 000	DeFi	DEX	Price Manipulation	L2
5	Fei Rari [129], [130]	2022-05-01	80 000 000	DeFi	Lending	Reentrancy	Mainnet
6	Infini [131], [132]	2025-02-24	49 500 000	DeFi	Stablecoin	Access Control	Mainnet
7	KyberSwap [133], [134]	2023-11-23	48 000 000	DeFi	DEX	Business Logic	Mainnet
8	Penpie [135], [136]	2024-09-03	27 000 000	DeFi	Yield Aggregator	Reentrancy	Mainnet
9	Sonne Finance [137], [138]	2024-05-16	20 000 000	DeFi	Lending	Rounding Error	L2
10	Inverse Fin. [139], [140], [141]	2022-04-02	15 600 000	DeFi	Lending	Price Manipulation	Mainnet
11	Holograph [142], [143]	2024-06-13	14 400 000	Token	NFT	Access Control	Mainnet
12	Deus DAO [144], [145]	2023-05-05	6 500 000	DeFi	Token	Access Control	L2
13	Ionic [146], [147]	2025-02-04	6 900 000	DeFi	Lending	Input Validation	L2
14	Ronin Bridge [148], [149]	2024-08-06	12 000 000	Bridge	Cross-chain Bridge	Access Control	Mainnet
15	LI.FI [150], [151], [152]	2024-07-16	11 600 000	DeFi	Cross-chain Bridge	Access Control	Mainnet
16	Yearn Finance [153], [154]	2023-04-13	11 400 000	DeFi	Yield Aggregator	Misconfiguration	Mainnet
17	zkLend [155], [156]	2025-02-12	9 600 000	DeFi	Lending	Rounding Error	L2
18	LI.FI Protocol [157], [158]	2024-07-16	11 600 000	DeFi	Cross-Chain	Unchecked Call Return Value, Access Control	Mainnet
19	Hundred Finance [159], [160]	2023-04-15	7 400 000	DeFi	Lending	Exchange Rate Manipulation, Rounding Error	L2
20	Abracadabra Money [161]	2024-01-31	6 500 000	DeFi	Lending	Business Logic, Rounding Error	Mainnet
21	Lodestar Finance [162], [163]	2022-12-10	6 500 000	DeFi	Lending	Price Manipulation	L2
22	IInch [164], [165]	2025-03-06	5 000 000	DeFi	DEX	Integer Underflow	Mainnet
23	Shezmu [166], [167]	2024-09-20	4 900 000	DeFi	Lending	Access Control	Mainnet
24	Gamma Strategies [168], [169]	2024-01-04	4 500 000	DeFi	Liquidity Pool	Price Manipulation	L2
25	Conic Finance [170], [171]	2023-07-25	4 200 000	DeFi	Lending	Read-Only Reentrancy, Price Manipulation	Mainnet
26	KiloEx [172], [173]	2025-04-17	7 500 000	DeFi	DEX	Access Control, Price Manipulation	L2
27	SIR Trading [174], [175]	2025-03-30	355 000	DeFi	Synthetics	Storage Collision	Mainnet
28	Abracadabra Money [176]	2025-03-26	13 000 000	DeFi	Lending	Business Logic	L2
29	DeltaPrime [177], [178]	2024-11-11	4 850 000	DeFi	Lending	Improper Input Validation	L2
30	Onyx Protocol [179], [180]	2024-09-26	3 800 000	DeFi	Lending	Exchange Rate Manipulation, Access Control	Mainnet
31	Dexible [181], [182]	2023-02-20	2 000 000	DeFi	DEX Aggregator	Access Control	Mainnet
32	Dough Finance [183], [184]	2024-07-12	2 100 000	DeFi	Lending	Improper Input Validation	Mainnet
33	CloverDEX [185], [186]	2024-12-10	501 279	DeFi	DEX	Reentrancy	L2
34	Tornado Cash [187], [188]	2023-05-20	2 173 500	DeFi	Privacy Mixer	Governance	Mainnet
35	Team Finance [189], [190]	2022-10-27	15 800 000	DeFi	Token Locker	Input Validation	Mainnet
36	Raft Protocol [191], [192]	2023-11-10	3 600 000	DeFi	Stablecoin	Rounding Error	Mainnet
37	Rho Market [193], [194]	2024-07-19	7 500 000	DeFi	Lending	Privileged Abuse, Price Manipulation	L2
38	UwuLend [195], [196]	2024-06-10	19 400 000	DeFi	Lending	Price Manipulation	Mainnet
39	Velocore [197], [198]	2024-06-02	6 800 000	DeFi	AMM	Underflow	L2
40	Curio [199], [200]	2024-03-23	16 000 000	DeFi	DAO voting system	Access Control, Dangerous Delegatecall	Mainnet
41	Unizen [201], [202]	2024-03-08	2 100 000	DeFi	DEX	Dangerous Delegatecall	Mainnet
42	Seneca Protocol [203], [204]	2024-02-28	6 400 000	DeFi	CDP	Input Validation	Mainnet
43	OKX DEX [205], [206]	2023-12-12	2 700 000	DeFi	DEX	Access Control, Dangerous delegatecall	Mainnet
44	Zunami Protocol [207], [208]	2023-08-13	2 100 000	DeFi	Stablecoin	Price Manipulation	Mainnet
45	EraLend [209], [210]	2023-07-25	3 400 000	DeFi	Lending	Read-Only Reentrancy, Price Manipulation	L2
46	Atlantis Loans [211], [212]	2023-06-10	2 500 000	DeFi	Lending	Governance	Sidechains
47	Sturdy Finance [213], [214]	2023-06-12	800 000	DeFi	Lending	Read-only Reentrancy, Price Manipulation	Mainnet
48	Jimbo's Protocol [215], [216]	2023-05-28	7 500 000	DeFi	Lending/DEX	Access Control, Price Manipulation	L2
49	Swaprum [217], [218]	2023-05-19	3 000 000	DeFi	DEX	Access Control	L2
50	Orion Protocol [219], [220]	2023-02-03	3 000 000	DeFi	DEX Aggregator	Reentrancy	Mainnet

by the compiler, turning a runtime vulnerability into a compile-time error. `public`, `private`, `internal`, or `external` visibility must be explicitly declared by functions.

4) **Suicidal Contract:** EVM's `selfdestruct` allows the contract to destroy itself and send the remaining Ether in the contract to a target address. Since the *Dencun* upgrade (EIP-6780, March 2024) `selfdestruct` no longer removes code or storage and no longer grants a gas refund, *unless it is invoked in the same transaction that deployed the contract* [228]. Outside that narrow case, it merely transfers the contract's Ether balance to the specified address, effectively disabling contract "suicide" and the attacks it enabled.

5) **Call-Stack Depth Limit:** The EVM caps a call chain at 1024 frames. Before EIP-150 [108] an attacker could, for about 40 000 gas, create 1024 nested calls and cause every subsequent `call` from a victim contract to fail (return 0). EIP-150 raised the sub-call base cost to 700 gas and, more importantly, restricted each call to forwarding at most 63/64 of its remaining gas. Because the available gas shrinks geometrically, filling the stack now exceeds the block gas limit, rendering the "stack-depth attack" economically infeasible. The depth cap nonetheless remains, so contracts should always check the return value (or use `try/catch`) when interacting with external contracts.

IV. HOW DO REAL-WORLD ATTACKS ON ETHEREUM SMART CONTRACTS ALIGN WITH ACADEMIC VULNERABILITY CLASSIFICATIONS?

To answer RQ2, we conducted a comprehensive analysis of 50 significant smart contract exploits that occurred between 2022 and early 2025. The collection, selection, and analysis protocol for these attacks is presented in Section II-B. Table V presents the main attributes of these attacks, including date, volume of financial loss, ecosystem, application type, and the exploited vulnerabilities (identified through manual expert analysis and based on active vulnerabilities in Section III-A). We present more information of each attack and its attributes (victim contact and attacker addresses, etc.) along with a summary of the technical details of each attack in our accompanying repository.

Our insights reveal that while real-world successful attacks involve technical weaknesses attributed to classes of known vulnerabilities (Section III-A), there are dominant patterns in the incidents that have been unnoticed by the literature. Access control vulnerabilities and price manipulation emerged as the most frequent and financially devastating attack vectors, accounting for a combined loss of \$697.7 million in our dataset (see Table VI). Access control-related incidents

TABLE VI: Breakdown of security vulnerabilities by frequency and financial impact across analyzed incidents

Vulnerability Category	Number of Incidents	Total Losses (USD)
Access Control	13	417,950,000
Price Manipulation	13	279,750,000
Reentrancy	7	118,501,279
Business Logic Flaws	5	177,400,000
Input Validation	6	41,850,000
Rounding Error	4	36,900,000
Integer Underflow	2	11,800,000
Governance	2	4,673,500
Dangerous Delegatecall	2	4,800,000
Storage Collisions	1	355,000

involve exploit methods beyond mere coding flaws (e.g., lack or misplacement of `require(msg.sender==owner)` checks) and include human-centric factors as well (see Insight 1). The high frequency of price manipulation attacks underscores the critical importance of secure, manipulation-resistant oracles (Insight 7). We also observed that while classic vulnerabilities like reentrancy persist, attackers are employing more sophisticated variants like *read-only reentrancy* and targeting new EVM features such as transient storage (Insight 9), opening up novel exploit avenues.

Incidents are rarely single-shot failures. Only 19 of the 50 exploits hinged on merely a code defect; the rest were *chains* of two or more weaknesses (Insight 3) or contained a human factor enabling the incident. For instance, governance mistakes (on a protocol design level) served as the entry point, which attackers then combined with a price-feed or arithmetic flaw to drain funds.

Human factors are involved. Roughly one-third of the breaches trace back to mis-set upgrade parameters, retained developer keys, or copy-paste mistakes/issues that automated analyses seldom flag (Insight 1). Where the literature focuses on the correctness of implementation, our data show that operational discipline and key management are at least as consequential.

Amplifiers and emergent surfaces. Flash-loan liquidity reliably magnified price manipulation attacks (Insight 5); protocol composability let a rounding bug in one contract propagate across lending markets (Insight 8); and brand-new EVM features such as transient storage introduced fresh attack vectors (Insight 9). These observations motivate the tiered root-cause model introduced later in Section V.

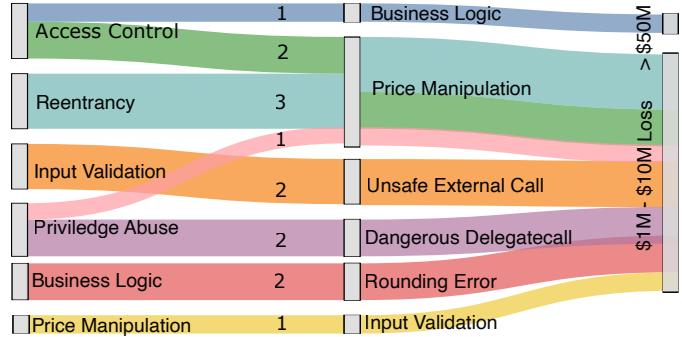
Insights 1–10 present the identified patterns in detail.

Insight 1: The Pervasiveness of Operational Human Mistakes

Human configuration errors manifest in ways such as: (1) *deployment/upgrade flaws*: we witnessed misconfigurations during upgrades (Nomad Bridge [26]), (2) incomplete upgrades (Ronin Bridge [148]), and bugs introduced in new code (LI.FI [150]). (3) *poor security hygiene*: retained privileges by former developers (Infini [132], Holograph [142]) and compromised private keys (OKX DEX [205], Rho Market [194]) led to significant losses. (4) *design oversights*: errors like the one in Yearn Finance [154] (incorrect address) or fundamental design flaws in logic or incentive mechanisms (Euler Finance’s liquidation discounts [25]) were exploited. (5) *ignoring warnings/known issues*: reusing code with known vulnerabilities or explicit warnings, as seen with EraLend [209], highlights a disregard for best practices.

Insight 2: Complex Cross-Chain Exploits

Three of the five largest losses (Nomad [26], Ronin [149], Rho Market [193]) exploited bridge or multi-chain logic, where state synchronicity is difficult to reason about formally. Emergence of these vulnerabilities calls for monitoring and analysis of cross-chain logic [229].



Incident Legend
 ● Euler Finance ● Hundred, Abracadabra ● KiloEx, Jimbo's ● Curio, OKX DEX
 ● LI.FI, Flexible ● Conic, EraLend, Sturdy ● Onyx Protocol ● Rho Market

Fig. 5: Incidents in which a chain of vulnerabilities was exploited. The first column (from left) represents the entry vulnerability used in an *exploit chain*, and the second column shows the second vulnerability abused in the exploit chain of the incident (number on the flow presents the number of incidents involving this exploit chain flow).

Insight 3: Multi-Vulnerability Exploit Chains

Many incidents were not caused by an isolated flaw but rather a combination of weaknesses (e.g., rows 1, 18, 19, 47, 48 in Table V). For example, the Sturdy Finance attack [213], [214] leveraged both read-only reentrancy and oracle manipulation to maximize extracted value, while the Jimbo protocol exploit [215], [216] combined price manipulation with access control flaws.

Euler Finance involved both access control and business logic flaws [123], [25], and LI.FI involved both access control and input validation issues [150], [151]. Figure 5 shows all of the incidents involving a chain of exploited vulnerabilities. As this figure shows, some vulnerabilities are only used to enable others to be exploited in an *exploit chain*. For instance, access control (Section III-A23) in all exploit chains (3 cases) is performed to pull off the second step of the attack (price manipulation and business logic). Furthermore, Figure 5 shows the role of reentrancy in exploit chains: as the thickest flow in the diagram shows, it is most commonly used to create an entry during which the attacker executes the second step: price manipulation. According to these results, in the case of an exploit chain, price manipulation is always enabled with one of the following vulnerabilities in the contract: access control, reentrancy, and privilege abuse (human factor).

Insight 4: Inadequate Emergency Response Mechanisms

The Seneca protocol incident, where pause functionalities were declared *internal*-only, rendered them useless during the attack. Such incident response functionalities are rarely used in practice, which might result in their incompetence [203].

Insight 5: Flash Loans as Attack Amplifiers

Flash loans were instrumental for price manipulation attacks (e.g., WooFi [128], Zunami Protocol [208]), reentrancy (Fei Rari [129]), and exploiting rounding errors (Sonne Finance [137]). They provide attackers with significant temporary capital to execute exploits that would otherwise be impractical.

Insight 6: Insider Threats and Privilege Abuse

Access control issues were not just the result of external attackers but also about insufficient organizational controls and excessive privileges (Swaprum [218]).

Insight 7: Oracle Security

The high number of price manipulation incidents underscores that secure, manipulation-resistant oracles are crucial. Reliance on spot prices from single DEXs, insufficient liquidity in oracle pairs, or oracles whose prices can be influenced within a single atomic transaction remains a major weak point.

Insight 8: Composability Risks

While DeFi's composability helps innovation, it also increases attack surfaces. Interactions between protocols (e.g., Abracadabra's cross-protocol state issue [176]) or complex internal logic can hide subtle bugs. Composability-aware analysis and verification of protocols helps prevent these issues [230].

Insight 9: New Attack Vectors for EVM Upgrades

The SIR Trading incident [175] which happened due to an attack vector on transient storage opcodes (EIP-1153 [231]) demonstrates that new EVM features can introduce unforeseen vulnerabilities if not implemented and understood correctly by developers.

Insight 10: Complications of Proxy/Upgradability Pattern

Proxy or upgradability patterns allow for bug fixes but also introduce severe risks if admin keys are compromised (in case of OKX DEX [205] incident) or if governance mechanisms controlling upgrades are subverted (Tornado Cash [187] and Atlantis Loans [212] incidents).

For a detailed review of each incident, refer to our accompanying repository³ providing in-depth analysis.

V. A MULTI-TIER ROOT-CAUSE FRAMEWORK FOR SMART CONTRACT EXPLOITS

We started by identifying smart contract vulnerability classes from academic literature (in Section III) to understanding which classes of these vulnerabilities are indeed exploited in real-world attacks (in Section IV). Our analysis of these high-impact incidents revealed that financially severe incidents often stem from sources beyond simple developer mistakes. These exploits frequently originate from flawed economic designs, broken operational procedures (e.g., governance protocols), and incorrect assumptions about a contract's environment. Based on the observations from the real-world incident analyses and our comprehensive SLR of vulnerability classifications, we propose a novel, *four-tier root-cause-based framework*.

The four-tier (not to be confused with blockchain layers) consist of: (1) *Flawed Economic Design & Protocol Logic*, where the core concept is unsound; (2) *Protocol Lifecycle & Governance Failures*, where human processes like upgrades, key management, or governance are broken; (3) *External Dependency Vulnerabilities*, where the protocol makes unsafe assumptions about external data or contracts; and (4) *Implementation-Level Weakness*, which encompasses classic coding errors. This model reveals that the most significant financial losses often stem from failures in the *first two levels, areas that code-centric audits may overlook*. This entails that in case of an existing flaw in an early tier (tier 1 and 2), the protocols will be exploitable regardless of quality of coding practices (lack of tiers 3 and 4 attack vectors). This model organizes exploit origins from the most fundamental (the design of the protocol) to the most specific (the implementation). It recognizes that a contract can be technically sound yet economically flawed, or developed securely from known smart contract vulnerabilities but managed insecurely. Table VII summarizes this framework, linking each category to the real-world incidents from our dataset of 50 attacks that exemplify it.

A. Tier 1: Flawed Economic Design & Protocol Logic

Here, the protocol is vulnerable not because of a bug, but because its core concept is economically or logically unsound (12 incidents trace back to these vulnerabilities). The code may execute exactly as intended, but the intended functionality itself is flawed. For example, the academic vulnerability of *business logic flaws* (where the contract does not match expected behavior) (Section III-A20) is a subset of this tier. However, our findings show this category is broader than business logic flaws, including the flawed liquidation incentive model in the Euler Finance [24] case and the insecure pricing formula in the WooFi [127] exploit, which are design-level errors, not simple implementation bugs.

B. Tier 2: Protocol Lifecycle & Governance Failures

Our empirical data reveals that a significant number of catastrophic incidents (12 incidents) originate from failures in the human and operational processes surrounding a smart contract. This tier is almost entirely absent from traditional, code-centric vulnerability taxonomies, yet it is a dominant cause of failure. Incidents like the Nomad Bridge exploit, caused by a clumsy upgrade procedure [26], or the Infini exploit, resulting from a failure to revoke developer privileges [132], would not have been detected by existing code analysis tools. They represent failures in deployment, maintenance, and governance procedures that show the need *to audit not just the code, but also the team's operational security*.

C. Tier 3: External Dependency Vulnerabilities

This tier addresses vulnerabilities and incidents arising from a contract's data dependencies with its environment (12 incidents). The contract's internal logic may be sound, but it makes unsafe assumptions about the inputs it receives from the outside world. This expands upon the literature's concept of *Untrustworthy Data Feeds* (Section III-A14) to include any external dependency. This includes not only price oracles [4], as seen in the Inverse Finance attack [233], but also unvalidated calldata from users, as exploited in the LI.FI incident [150]. Furthermore, our analysis identifies the critical role of environmental factors like the availability of *flash loans, which, while not a vulnerability themselves, act as powerful amplifiers for exploits in this tier*.

D. Tier 4: Implementation-Level Weakness

This tier aligns most closely with the traditional vulnerability classifications identified in our SLR (RQ1 results in Section III). We traced back the origin of 14 incidents to these vulnerabilities (Table VII). These are concrete programming mistakes where the contract contains a known vulnerability. This includes well-documented issues such as reentrancy (Section III-A3), as seen in many attacks such as Fei Rari exploit [234], and integer over/underflow (Section III-A1), found in multiple incidents such as Velocore [197]. While critically important, the number of incidents that their root-cause is attributed to other tiers (see Table VII) suggests an incomplete picture of real-world attacks solely based on tier 4 vulnerabilities.

E. A Case Study for Tier 2

In order to investigate the usefulness of the framework proposed, we analyze "clumsy deployments and upgrades"—a sub-category of tier 2 vulnerabilities (from Table VII). This sub-category consists of contract upgrades and initialization that lead to vulnerabilities. Four of the largest bridge, vault, and DEX catastrophes in our dataset (IDs 2, 14, 16, 41; \$215.5M combined loss) share the *same* root-cause: a seemingly flawless deployment and upgrade that left the contract in an unsafe state while the byte-code itself remained bug-free. Treating them jointly shows that our sub-category is the *minimal sufficient grain* at which attacker assumptions, exploit steps, and defenses can be specified once and reused to prevent future exploits to similar protocols.

³<https://github.com/HadiRe/SoK-Root-Cause-of-Smart-Contract-Incidents>

TABLE VII: A multi-tier root-cause framework derived from real-world incidents. Incident IDs refer to rows in Table V.

Tier	Sub-Category	Description ← Incidents Involved (ID)
T1: Flawed Economic Design & Protocol Logic	Incorrect Pricing/Valuation Models	The protocol's core formulas for calculating value are inherently manipulable. <i>Incidents:</i> 4, 7, 19, 24, 30, 44, 48
	State Inconsistency & Desynchronization	The logic allows different parts of the protocol to hold contradictory state information, enabling exploits. <i>Incidents:</i> 20, 28, 45, 47
	Perverse Incentive Mechanisms	The design unintentionally rewards behavior that harms the protocol, such as self-liquidation. <i>Incidents:</i> 1
T2: Protocol Lifecycle & Governance Failures	Clumsy Deployments & Upgrades	Mistakes during contract initialization or upgrades create vulnerabilities. <i>Incidents:</i> 2, 14, 16, 41
	Compromised Keys & Insider Threats	Administrative keys are stolen or abused by internal actors (current or former) to bypass security. <i>Incidents:</i> 6, 11, 37, 43, 49
	Governance System Exploitation	The rules of the DAO are manipulated to pass malicious proposals or seize administrative control. <i>Incidents:</i> 34, 40, 46
T3: External Dependency Vulnerabilities	Oracle & Price Feed Manipulation	The protocol uncritically trusts an external price feed that is susceptible to manipulation. <i>Incidents:</i> 3, 10, 21, 38
	Unvalidated User/Contract Inputs	The protocol fails to sanitize or validate data and external calls originating from untrusted users. <i>Incidents:</i> 13, 15, 18, 29, 31, 32, 35, 42
T4: Impl.-Level Vulnerabilities	Interaction Pattern Failures	The code violates secure interaction patterns (e.g., CEI), enabling reentrancy. <i>Incidents:</i> 5, 8, 25, 33, 50
	Arithmetic Errors	The code is susceptible to over/underflow, or precision loss due to EVM integer math. <i>Incidents:</i> 9, 17, 22, 36, 39
	Flawed Access Control Logic	A flaw in the access control implementation (e.g., reversed parameters) allows it to be bypassed. <i>Incidents:</i> 12, 23, 26
	State & Data Handling Errors	The code incorrectly manages data representation or state, leading to issues like storage collisions. <i>Incidents:</i> 27

TABLE VIII: Defenses that neutralise four incidents in “clumsy deployments & upgrades” sub-category.

Defense	Example	Effect on Incidents (2/14/16/41)
Invariant-checked upgrades	Foundry[232]/Echidna[9] test asserting post-upgrade state ($\neq 0$, correct addresses, guard present).	Blocks deployment; test fails on first run.
Staged roll-outs	Deploy first on canary fork → 24h observation → mainnet.	Limits loss to canary TVL.
Operational gates	Multi-sig + two-step pause → upgrade → un-pause.	Human review spots violating states.
Runtime monitors	Circuit-breaker on abnormal withdrawals / mint rates / external calls.	Halts draining within the first block.

1) Attacker Model: Based on our incident analyses, we envision an attacker with the following attributes:

Goal: Extract maximum value by exploiting mis-configured contract state after upgrade or deployment.

Capabilities: i) Craft arbitrary on-chain transactions (privileged keys not required). ii) Inspect contract storage. iii) Use flash loans to amplify the effect. iv) Reads the verified source code of the protocol/ABI.

Aussumption: Upgrades already live as contracts on mainnet.

2) **Exploit Walk-Through:** The incidents can be broken down into three steps:

Step A (State discovery): Attacker reads mis-initialised variable (Nomad committedRoot, Ronin _totalOperatorWeight, Yearn vault basket, Unizen call-guard) and confirms it lifts a safety check.

Step B (Trigger): Crafts transaction exploiting the lifted check: replay any root (Nomad), sign with tiny weight (Ronin), mint vault shares (Yearn), or route trade through the attacker contract (Unizen).

Step C (Extraction): Drains bridge funds / swaps inflated shares / spends user approvals. No reentrancy or oracle tricks are needed; the upgrade itself is the point of failure.

3) **Defense Model:** Table VIII presents our proposed defenses against the attacker model of this sub-category. Before moving real assets, effective tier 2 security must wrap the *entire upgrade life-cycle*: (i) *pre-deployment/upgrade* initiation property-based tests that abort if critical state is mis-initialised; (ii) a *canary rollout* that risks only a capped pilot TVL while the new code faces live mainnet conditions for 24h; (iii) human *multi-sig gating* that forces a pause-upgrade-unpause ritual; and (iv) on-chain *circuit-breakers* [43] that freeze abnormal withdrawals, mint rates, or delegate-calls within the first block.

VI. CONCLUSION AND FUTURE WORK

Threats to Validity. Our evidence base contains only incidents that were *publicly revealed* and *financially substantial*, so *silent failures* and *minor incidents* remained outside the scope of this work which can affect our results. All data came from the EVM blockchain

family, which could lead to an alternative execution platform exposing additional incident patterns not observed in our study.

Future Directions. At the economic layer, the community needs agent-based simulation, model-checking, and other formal-methods machinery that can prove the designed system preserves economic soundness. At the governance and lifecycle tier, development pipelines should integrate upgrade simulators, proposal-linting tools and role-minimisation checks. External data dependency analysis can evolve into *whole-mempool* and *cross-protocol reasoning* that surfaces oracle manipulation or flash-loan amplification before execution. Running across all tiers, real-time monitoring observing mempool based on exploit-chain-aware rules should be investigated. Finally, as bridges and roll-ups are common among loss statistics (Table V), they require more investigation as the contracts interacting are heterogeneous [229].

Conclusion. This study unites the literature’s focus on implementation bugs with the practitioner’s day-to-day experience of successful attacks on Ethereum-based protocols. Through a systematic literature review of 71 peer-reviewed papers and dissecting 50 publicly disclosed exploits that caused a total loss of \$1 093 979 779 between 2022 and 2025, we show that less than one-third of recorded losses can be traced to an isolated coding mistake. The remainder cascaded from flawed incentive design, clumsy governance decisions, unsafe assumptions about external dependencies, or exploit chains-factors that mainstream static and dynamic analyzers rarely take into account. To capture this, we introduced a four-tier root-cause framework ranging from economic logic at tier 1 to programming mistakes at tier 4; together with our labeled incident dataset and mitigation catalog, the framework equips auditors, tool builders and protocol designers with a map of the modern threat surface based on real incidents.

APPENDIX

REFERENCES

- [1] Coinbase, “Ethereum Price, Charts, and News.” <https://www.coinbase.com/price/ethereum>. 2025-04-19.

TABLE A.1: Mapping of exploit transaction hashes, compromised victim contracts, and attacker addresses (smart contracts and EOAs).

ID	Attack Transactions	Victim Contract	Attacker Smart Contract/EOA
1	0xc310, 0x62bd, 0x3097, 0x47ac	0xe025	0xebc2, 0x036c, 0xd3b7, 0x0b81
2	0xa5fe	0x88a6	0xb5c5
3	0x3195	0x8f55	0xcaf
4	0xe80a	0xeff2	0x9961
5	0xd9ee, 0xadbe, 0xf75, 0x0742, 0x2547, 0xab48	0x3207	0xE39f, 0x3207
6	0xa5fe	0x9a79	0xc49b
7	0x485e, 0x09a3, 0x396a	0xaf2a	0x5027, 0xc9b8
8	0x7c7f, 0xfd0, 0x7961, 0xca87	-	0x7a2f
9	0x9312	sovelo, sousdc, soweth	0x02fa
10	0x561e, 0x20a6, 0x6003	0x39b1	0x117c, 0x8b4c
11	0x0cc1	-	-
12	0x6129	0xbc1b	0xe2ee
13	0x9aa3, 0x37e5, 0x5db6	0x964d	0x9e34
14	0x2619	0xfc27	-
15	0xd82f, 0x65a9	0xf28a	0x7742
16	0xd55e, 0x8db0	-	0x5bac, 0x16af
17	0x0160	0x04c0	0x04d7
18	0xd82f, 0x65a9	0xc74f	0x7742
19	0x6e9e, 0x1509	0x8adb	0x155d
20	0x26a8, 0xdb46	0x7259	0x87f5
21	0xc523	0x7596	0xc29d
22	0x0497, 0xb5c9, 0xb16b, 0x3947, 0x9ce5, 0xb068, 0x6273	0xb02f	0x019b
23	0x3932	0xa35f	0xa3a6
24	0x025c	0x4b57	0x5351
25	0x8b74	0x8d67	0x10db
26	0x6b37	0x551f	0x00fa
27	0xa05f	0xb91a	0x27de
28	0xed17	0x625F	0xa9e
29	0x9efe	0x52ee	0xb878
30	0x4656	-	0x6809
31	0x138d	0xde62	0x6840
32	0x92cd	0x534a, 0x53d2, 0x0d2b, 0x8f69, 0xc581, 0x460f, 0x3e9f, 0x52f0	0x6710
33	0x8fc0	0x6a0b	0x012f
34	0x3274	0xc503	0x0921 , 0x5923
35	0xb2e3	0cff0	0cff0
36	0xfeed	0x9ab6, 0xd0db	0xa033, 0xfd0, 0x0119
37	-	0x2a9c	0x893d
38	0x242a0f, 0xb3f0, 0xca1b	-	0x841d
39	0xed11d, 0x3743, 0x4156	0xed4e , 0xb7f6 , 0xc030, 0xf7f7	0x8cdc
40	0x4ff40	-	0xdaaa
41	0x30fe, 0xc12a	0xd3f6	0xb660,0x90a7, 0x2f74
42	0xee02	-	0x9464, 0x5217,0xe83b
43	0xc6a5, 0x22eb	0x70cb , 0x40aa , 0x55b3	0xfacf
44	0x0788, 0x2aec	0xb40b	0x5f4c
45	0x7ac4, 0x99ef	0x7d87, 0xd5c6	0xf1d0
46	0xeade	0x613c	0xeade
47	0xeb87	0x0b09	0x1e84
48	0x44a0	0x2719	0x102b
49	0x36fe	0x9980	0xf274
50	0xa6f6	0xb559	0x5061

- [2] V. Buterin *et al.*, “A Next-Generation Smart Contract and Decentralized Application Platform,” *white paper*, vol. 3, no. 37, pp. 2–1, 2014.
- [3] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, “Sok: Decentralized finance (defi) attacks,” in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2444–2461, IEEE, 2023.
- [4] M. Eshghie, M. Jafari, and C. Artho, “From Creation to Exploitation: The Oracle Lifecycle,” in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, pp. 23–34, Mar. 2024.
- [5] Chainalysis, “\$2.2 Billion Stolen from Crypto Platforms in 2024,” <https://www.chainalysis.com/blog/crypto-hacking-stolen-funds-2025/>. 2025-06-19.
- [6] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 254–269, 2016.
- [7] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (sok),” in *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings* 6, pp. 164–186, Springer, 2017.
- [8] J. Feist, G. Grieco, and A. Groce, “Slither: a static analysis framework for smart contracts,” in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WET-SEB)*, pp. 8–15, IEEE, 2019.
- [9] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, “Echidna: effective, usable, and fast fuzzing for smart contracts,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020*, p. 557–560, ACM, 2020.
- [10] M. di Angelo, T. Durieux, J. a. F. Ferreira, and G. Salzer, “SmartBugs 2.0: An Execution Framework for Weakness Detection in Ethereum Smart Contracts,” in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering, ASE ’23*, p. 2102–2105, IEEE Press, 2024.
- [11] T. Durieux, J. a. F. Ferreira, R. Abreu, and P. Cruz, “Empirical review of automated analysis tools on 47,587 ethereum smart contracts,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE ’20*, p. 530–541, ACM, 2020.
- [12] H. Chen, M. Pendleton, L. Njilla, and S. Xu, “A survey on ethereum systems security: Vulnerabilities, attacks, and defenses,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–43, 2020.

- [13] G. de Sousa Matsumura, L. B. R. dos Santos, A. F. d. Conceição, and N. L. Vijaykumar, “Vulnerabilities and open issues of smart contracts: A systematic mapping,” in *International Conference on Computational Science and Its Applications*, pp. 3–18, Springer, 2021.
- [14] N. F. Samreen and M. H. Alalfi, “A survey of security vulnerabilities in ethereum smart contracts,” in *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering, CASCON ’20*, (USA), p. 73–82, IBM Corp., 2020.
- [15] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, “A survey of smart contract formal specification and verification,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 7, pp. 1–38, 2021.
- [16] Y. Kissoon and G. Bekaroo, “Detecting vulnerabilities in smart contract within blockchain: a review and comparative analysis of key approaches,” in *2022 3rd International Conference on Next Generation Computing Applications (NextComp)*, pp. 1–6, IEEE, 2022.
- [17] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, “Systematic review of security vulnerabilities in ethereum blockchain smart contract,” *IEEE Access*, vol. 10, pp. 6605–6621, 2022.
- [18] 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*, p. 107221, 2023.
- [19] S. Porkodi and D. Kesavaraja, “Smart contract: A survey towards extortive vulnerability detection and security enhancement,” *Wireless Networks*, vol. 30, no. 3, pp. 1285–1304, 2024.
- [20] F. Vidal, N. Ivaki, and N. Laranjeiro, “Opencsv: an open hierarchical taxonomy for smart contract vulnerabilities,” *Empirical Software Engineering*, vol. 29, no. 4, pp. 1–47, 2024.
- [21] C. Ruggiero, P. Mazzini, E. Coppa, S. Lenti, and S. Bonomi, “Sok: A unified data model for smart contract vulnerability taxonomies,” in *Proceedings of the 19th International Conference on Availability, Reliability and Security (ARES 2024)*, pp. 1–11, ACM, 2024.
- [22] A. Augusto, R. Belchior, M. Correia, A. Vasconcelos, L. Zhang, and T. Hardjono, “Sok: Security and privacy of blockchain interoperability,” in *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 3840–3865, IEEE, 2024.
- [23] M. J. Islam, S. Islam, M. Hossain, S. Noor, and S. R. Islam, “Securing blockchain systems: A layer-oriented survey of threats, vulnerability taxonomy, and detection methods,” *Future Internet*, vol. 17, no. 5, p. 205, 2025.
- [24] Rekt News, “Euler REKT.” <https://rekt.news/euler-rekt>. 2025-04-15.
- [25] SlowMist, “SlowMist: An Analysis of the Attack on Euler Finance.” <https://medium.com/p/5143abc0d5ad>. 2025-03-29.
- [26] Rekt News, “Nomad bridge.” <https://rekt.news/nomad-rekt>. 2022-08.
- [27] B. Kitchenham, S. Charters, et al., “Guidelines for performing systematic literature reviews in software engineering version 2.3,” *Engineering*, vol. 45, no. 4ve, p. 1051, 2007.
- [28] Rekt News, “Rekt News Leaderboard.” <https://rekt.news/leaderboard/>. 2025-05-08.
- [29] DeFiYield Team, “DeFiYield REKT Database.” <https://defiyield.app/rekt-database>. 2025-05-08.
- [30] SlowMist Team, “SlowMist Hacked Database.” <https://hacked.slowmist.io/>. 2025-05-08.
- [31] BlockSec Team, “BlockSec Blog on Medium.” <https://blocksecteam.medium.com/>. 2025-05-08.
- [32] Base Documentation Team, “Bridges on base mainnet.” <https://docs.base.org/chain/bridges-mainnet>. 2025-05-18.
- [33] Ethereum Foundation, “Ethereum scaling.” <https://ethereum.org/en/developers/docs/scaling>. 2025-05-18.
- [34] C. F. Torres, J. Schütte, and R. State, “Osiris: Hunting for integer bugs in ethereum smart contracts,” in *Proceedings of the 34th annual computer security applications conference*, pp. 664–676, 2018.
- [35] OpenZeppelin, “Openzeppelin/openzeppelin-solidity.” <https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol>. 2025-06-12.
- [36] S. Documentation, “Security considerations.” <https://docs.soliditylang.org/en/latest/security-considerations.html>. 2025-03-18.
- [37] Uniswap, “FullMath UniSwap.” <https://docs.uniswap.org/contracts/v3/reference/core/libraries/FullMath>. 2025-07-18.
- [38] 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.
- [39] A. Mense and M. Flatscher, “Security vulnerabilities in ethereum smart contracts,” in *Proceedings of the 20th international conference on information integration and web-based applications & services*, pp. 375–380, 2018.
- [40] E. Foundation, “Solidity documentation: Control structures.” <https://docs.soliditylang.org/en/latest/control-structures.html>. 2025-02-24.
- [41] Y. Zhang, J. Ma, X. Liu, G. Ye, Q. Jin, J. Ma, and Q. Zhou, “An efficient smart contract vulnerability detector based on semantic contract graphs using approximate graph matching,” *IEEE Internet of Things Journal*, 2023.
- [42] F. Vollero, “Checks-effects-interactions pattern.” https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html. 2025-03-04.
- [43] M. Eshghie, C. Artho, W. Ahrendt, T. Hildebrandt, and G. Schneider, “Capturing smart contract design with dcr graphs,” in *Software Engineering and Formal Methods*, vol. 14001 of *Lecture Notes in Computer Science*, pp. 116–134, Springer, 2023. https://link.springer.com/chapter/10.1007/978-3-031-47115-5_7.
- [44] L. Zhang, Y. Li, R. Guo, G. Wang, J. Qiu, S. Su, Y. Liu, G. Xu, H. Chen, and Z. Tian, “A novel smart contract reentrancy vulnerability detection model based on bigas,” *Journal of Signal Processing Systems*, pp. 1–23, 2023.
- [45] B. Jiang, Y. Liu, and W. K. Chan, “Contractfuzzer: Fuzzing smart contracts for vulnerability detection,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 259–269, 2018.
- [46] Ethereum Foundation, “ERC-2771: Secure Protocol for Native Meta Transactions.” <https://eips.ethereum.org/EIPS/eip-2771>. 2025-01-02.
- [47] OpenZeppelin, “Erc2771context multcall vulnerability disclosure.” <https://blog.openzeppelin.com/arbitrary-address-spoofing-vulnerability-erc2771context-multcall-public-disclosure>. 2025-01-02.
- [48] Shashank, “Unveiling the ERC-2771 Context and Multicall Vulnerability.” <https://blog.solidityscan.com/unveiling-the-erc-2771context-a-and-multicall-vulnerability-f96ffa5b49ff>. 2025-01-02.
- [49] Solidity Documentation, “Solidity v0.8.0: Libraries.” <https://docs.soliditylang.org/en/v0.8.0/contracts.html#libraries>. 2025-02-26.
- [50] SWC Registry, “SWC-112: Delegatecall to Untrusted Callee.” <https://swcregistry.io/docs/SWC-112>. 2025-02-26.
- [51] M. Eshghie, V. Åryd, C. Artho, and M. Monperrus, “Solidiffy: AST differencing for solidity smart contracts,” *arXiv preprint arXiv:2411.07718*, 2025.
- [52] Nick Mudge, “EIP-2535: Diamond Standard (Final).” <https://eips.ethereum.org/EIPS/eip-2535>. 2025-02-26.
- [53] OpenZeppelin, “UUPS Proxies.” <https://docs.openzeppelin.com/contracts/4.x/api/proxy#UUPSUpgradeable>. 2025-02-26.
- [54] OpenZeppelin, “Address utility.” <https://docs.openzeppelin.com/contracts/5.x/api/utils#Address>. 2025-02-18.
- [55] A. Ghaleb, J. Rubin, and K. Pattabiraman, “Etainter: detecting gas-related vulnerabilities in smart contracts,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 728–739, 2022.
- [56] RareSkills, “The rareskills book of solidity gas optimization: 80+ tips.” <https://www.rareskills.io/post/gas-optimization>. 2025-02-18.
- [57] Hacken, “Solidity gas optimization: Best practices & expert tips.” <https://hacken.io/discover/solidity-gas-optimization/>. 2025-02-17.
- [58] J. Su, H.-N. Dai, L. Zhao, Z. Zheng, and X. Luo, “Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–12, 2022.
- [59] Infuy, “Preventing denial of service attacks in solidity.” <https://www.infuy.com/blog/preventing-denial-of-service-attacks-in-solidity/>. 2025-04-18.
- [60] Z. Liao, S. Hao, Y. Nan, and Z. Zheng, “Smartstate: Detecting state-reverting vulnerabilities in smart contracts via fine-grained state-dependency analysis,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 980–991, 2023.
- [61] D. He, R. Wu, X. Li, S. Chan, and M. Guizani, “Detection of vulnerabilities of blockchain smart contracts,” *IEEE Internet of Things Journal*, 2023.
- [62] QuickNode, “Common solidity vulnerabilities on ethereum.” <https://www.quicknode.com/guides/ethereum-development/smart-contracts/common-solidity-vulnerabilities-on-ethereum>. 2025-03-24.
- [63] A. Reutov, “Predicting random numbers in ethereum smart contracts,” *Medium*, 2018.
- [64] OWASP Foundation, “OWASP Smart Contract Top 10.” <https://owasp.org/www-project-smart-contract-top-10/>. 2025-03-03.

- [65] 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.
- [66] Chainlink, “Secure randomness in solidity smart contracts.” <https://bl.og.chain.link/random-number-generation-solidity/>. 2025-01-18.
- [67] Chainlink, “Chainlink vrf: Verifiable randomness.” <https://docs.chain.link/vrf/v2.5>. 2025-04-15.
- [68] S. Ahmadjee, C. Mera-Gómez, S. Farshidi, R. Bahsoon, and R. Kazman, “Decision support model for selecting the optimal blockchain oracle platform: An evaluation of key factors,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 1, pp. 1–35, 2025.
- [69] E. Foundation, “Commit-reveal scheme in solidity.” <https://ethereum.org/en/developers/docs/commit-reveal/>. 2023.
- [70] Block.one, “Eos.io technical white paper.” <https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md>. 2025-03-08.
- [71] Ethereum Foundation, “Btcrelay: A bitcoin relay for ethereum smart contracts.” <https://github.com/ethereum/btcrelay/blob/develop/README.md>. 2025-04-13.
- [72] Infuy, “Understanding phishing with tx.origin in solidity.” <https://www.infuy.com/blog/understanding-phishing-with-tx-origin-in-solidity/>. 2025-02-23.
- [73] SlowMist, “Common vulnerabilities in solidity: Phishing with tx.origin.” <https://www.slowmist.com/articles/solidity-security/Common-Vulnerabilities-in-Solidity-Phishing-with-tx.origin.html>. 2025-04-05.
- [74] W. Zhang, Z. Zhang, Q. Shi, L. Liu, L. Wei, Y. Liu, X. Zhang, and S.-C. Cheung, “Nyx: Detecting exploitable front-running vulnerabilities in smart contracts,” in *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 2198–2216, IEEE, 2024.
- [75] SCSFG, “Hackers’ guide to front-running in blockchain.” <https://scsf.g.io/hackers/frontrunning/>. 2025-02-18.
- [76] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding the greedy, prodigal, and suicidal contracts at scale,” in *Proceedings of the 34th annual computer security applications conference*, pp. 653–663, 2018.
- [77] E. Foundation, “Smart contract security guidelines.” <https://ethereum.org/en/developers/docs/smart-contracts/security/>. 2025-02-27.
- [78] Uniswap, “Uniswap v3 twap documentation.” <https://docs.uniswap.org/concepts/V3-overview/oracle>. 2024-10-05.
- [79] S. Wu, Z. Yu, D. Wang, Y. Zhou, L. Wu, H. Wang, and X. Yuan, “Defiranger: Detecting defi price manipulation attacks,” *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 4147–4161, 2024.
- [80] Tellor, “Tellor, a community, an oracle, unstoppable.” <https://tellor.io/whitepaper/>. 2023-10-20.
- [81] Z. Chen, Y. Liu, S. M. Beillahi, Y. Li, and F. Long, “Demystifying invariant effectiveness for securing smart contracts,” *Proc. ACM Softw. Eng.*, vol. 1, July 2024.
- [82] M. Xie, M. Hu, Z. Kong, C. Zhang, Y. Feng, H. Wang, Y. Xue, H. Zhang, Y. Liu, and Y. Liu, “Defort: Automatic detection and analysis of price manipulation attacks in defi applications,” *ISSTA 2024*, p. 402–414, ACM, 2024.
- [83] J. Zhong, D. Wu, Y. Liu, M. Xie, Y. Liu, Y. Li, and N. Liu, “Defiscope: Detecting various defi price manipulations with LLM reasoning,” *CoRR*, vol. abs/2502.11521, 2025.
- [84] A. Alhaidari, B. Palanisamy, and P. Krishnamurthy, “Flashguard: Real-time disruption of non-price flash loan attacks in defi,” in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS ’24*, p. 5039–5041, ACM, 2024.
- [85] M. Demir, M. Alalfi, O. Turetken, and A. Ferworn, “Security smells in smart contracts,” in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 442–449, IEEE, 2019.
- [86] RagHAV, “Duaraghav8/Ethlint.” <https://github.com/duaraghav8/Ethlint>. 2025-03-02.
- [87] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, “Exploiting the laws of order in smart contracts,” in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pp. 363–373, 2019.
- [88] S. Hwang and S. Ryu, “Gap between theory and practice: An empirical study of security patches in solidity,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 542–553, 2020.
- [89] E. Foundation, “Solidity documentation - type conversions.” <https://docs.soliditylang.org/en/latest/types.html#explicit-conversions>. 2024-03-09.
- [90] Solidity Team, “Types — solidity documentation.” <https://docs.soliditylang.org/en/latest/types.html>. 2024-10-20.
- [91] S. Team, “Latest solidity release and updates.” <https://github.com/ethereum/solidity/releases>. 2025-03-09.
- [92] Z. Zheng, W. Chen, Z. Zhong, Z. Chen, and Y. Lu, “Securing the ethereum from smart ponzi schemes: Identification using static features,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–28, 2023.
- [93] Investopedia, “Ponzi schemes in cryptocurrency: What you need to know.” <https://www.investopedia.com/terms/p/ponzi-scheme.asp>. 2025-04-24.
- [94] N. Ruaro, F. Gritti, R. McLaughlin, I. Grishchenko, C. Kruegel, and G. Vigna, “Not your type! detecting storage collision vulnerabilities in ethereum smart contracts,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, pp. 1–17, 2024.
- [95] S. Documentation, “View functions in solidity.” <https://docs.soliditylang.org/en/latest/contracts.html#view-functions>. 2025-01-26.
- [96] V. Wüstholtz and M. Christakis, “Harvey: A greybox fuzzer for smart contracts,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1398–1409, 2020.
- [97] S. Registry, “Write to arbitrary storage location.” <https://swcregistry.io/docs/SWC-124/>. 2025-02-08.
- [98] OpenZeppelin, “Proxy upgrade pattern.” <https://docs.openzeppelin.com/upgrades-plugins/proxies>. 2025-02-02.
- [99] S. Yao, H. Ni, A. C. Myers, and E. Cecchetti, “Scif: A language for compositional smart contract security,” *arXiv preprint arXiv:2407.01204*, 2024.
- [100] Chainlight, “Avoiding storage collisions in solidity smart contracts.” <https://blog.chainlight.io/avoiding-storage-collisions-in-solidity>. 2025-04-02.
- [101] B. Zhang, “Towards finding accounting errors in smart contracts,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.
- [102] Z. Zhuo and Web3Bugs, “Web3bugs - smart contract vulnerabilities.” <https://github.com/ZhangZhuoSJTU/Web3Bugs/tree/main/contracts/16>. 2025-02-03.
- [103] M. Eshghie and C. Artho, “Oracle-Guided Vulnerability Diversity and Exploit Synthesis of Smart Contracts Using LLMs,” in *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 2240–2244, Oct. 2024.
- [104] Y. Liu, Y. Li, S.-W. Lin, and Q. Yan, “ModCon: A model-based testing platform for smart contracts,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020*, pp. 1601–1605, ACM, Nov. 2020.
- [105] G. Morello, M. Eshghie, S. Bobadilla, and M. Monperrus, “Disl: Fueling research with a large dataset of solidity smart contracts.” <https://arxiv.org/abs/2403.16861>.
- [106] S. Documentation, “Assertions and error handling in solidity.” <https://docs.soliditylang.org/en/latest/control-structures.html#assert>. 2025-04-16.
- [107] M. Eshghie, G. Andersson Kasche, C. Artho, and M. Monperrus, “SInDi: Semantic Invariant Differencing for Solidity Smart Contracts,” technical report, KTH Royal Institute of Technology, Theoretical Computer Science Division, 2025. Published via KTH DiVA Portal.
- [108] E. Foundation, “Eip-150: Gas cost changes for io-heavy operations.” <https://eips.ethereum.org/EIPS/eip-150>. 2025-04-21.
- [109] D. Perez and B. Livshits, “Broken metre: Attacking resource metering in evm,” in *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA, USA), Internet Society, February 2020.
- [110] S. V. Database, “High severity (7.5) resource exhaustion in github.com/ethereum/go-ethereum/eth/gasprice.” <https://security.snyk.io/vuln/SNYK-GOLANG-GITHUBCOMETHEREUMGOETHERUMETHGASPRICE-6810217>. 2025-02-13.
- [111] Go-Ethereum, “internal/ethapi: crash caused by no limits for reward-percentiles at interface feehistory.” <https://github.com/ethereum/go-ethereum/issues/29709>. 2025-02-13.
- [112] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, “Finding permission bugs in smart contracts with role mining,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022*, pp. 716–727, ACM, July 2022.

- [113] H. H. Nguyen, N.-M. Nguyen, H.-P. Doan, Z. Ahmadi, T.-N. Doan, and L. Jiang, "Mando-guru: vulnerability detection for smart contract source code by heterogeneous graph embeddings," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1736–1740, 2022.
- [114] M. Rossini, M. Zichichi, and S. Ferretti, "On the use of deep neural networks for security vulnerabilities detection in smart contracts," in *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 74–79, IEEE, 2023.
- [115] Z. Wang, Q. Zheng, and Y. Sun, "Gvd-net: Graph embedding-based machine learning model for smart contract vulnerability detection," in *2022 International Conference on Algorithms, Data Mining, and Information Technology (ADMIT)*, pp. 99–103, IEEE, 2022.
- [116] OpenZeppelin, "Ownable - openzeppelin contracts." <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol>. 2023-09-05.
- [117] OpenZeppelin, "Accesscontrol - openzeppelin contracts." <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol>. 2025-02-23.
- [118] Halborn, "What is timestamp dependence??" <https://www.halborn.com/blog/post/what-is-timestamp-dependence>. 2025-03-07.
- [119] G. S. World, "What is timestamp dependence vulnerability??" <https://www.getsecureworld.com/blog/what-is-timestamp-dependence-vulnerability>. 2023.
- [120] O. Forum, "Using block number vs block timestamp in governor.sol." <https://forum.openzeppelin.com/t/using-block-number-vs-block-timesamp-in-governor-sol/30397>. 2025-04-20.
- [121] GloryPraise, "Vulnerability 8: Timestamp dependence vulnerability." <https://glorypraise.hashnode.dev/vulnerability-8-timestamp-dependence-vulnerability>. 2025-03-07.
- [122] Cobalt, "Cobalt blog." <https://www.cobalt.io/blog>. 2025-03-07.
- [123] Immunebytes, "Euler Finance Hack - Mar 13, 2023: Detailed Hack Analysis." <https://immunebytes.com/blog/euler-finance-hack-mar-1-3-2023-detailed-hack-analysis/>. 2025-03-29.
- [124] Rekt, "Euler finance." <https://rekt.news/zh/euler-rekt>. 2025-04-15.
- [125] Nomad XYZ, "Nomad Bridge Hack Root Cause Analysis." <https://medium.com/p/875ad2e5aacd>. 2025-03-29.
- [126] R. News, "Rekt news - defi investigations and analysis." <https://rekt.news/bonq-rekt>. 2025-03-29.
- [127] Rekt News, "Woofi." <https://rekt.news/woo-rekt>. 2025-04-15.
- [128] Olympix, "Unpacking the woofi swap exploit." <https://medium.com/p/ae6f172fe736>. 2025-04-15.
- [129] R. News, "Fei rari." <https://rekt.news/fei-rari-rekt>. 2025-03-29.
- [130] CertiK, "Fei protocol incident analysis." <https://medium.com/p/8527440696cc>. 2025-04-13.
- [131] R. News, "Infini." <https://rekt.news/infini-rekt>. 2025-04-13.
- [132] G. Yona, "Rogue developer suspected in infini neobank's \$49.5m exploit." <https://news.shib.io/2025/02/24/rogue-developer-suspected-in-infini-neobanks-49-5m-exploit/>. 2025-04-26.
- [133] REKT, "Kyberswap." 2023-11-23.
- [134] SlowMist, "A deep dive into the kyberswap hack." <https://medium.com/p/3e13f3305d3a>.
- [135] Olympix, "Penpie exploit: A technical post-mortem analysis," 2024-09-06.
- [136] SolidityScan, "Penpie hack analysis," 2024-09-05.
- [137] CertiK, "Sonne finance incident analysis." <https://rekt.news/sonne-finance-rekt>. 2025-03-01.
- [138] R. News, "Curve, vyper." <https://rekt.news/curve-vyper-rekt>.
- [139] REKT, "Inverse finance." <https://rekt.news/inverse-finance-rekt/>. 2022-04-03.
- [140] Cycle Trading, "Inverse finance: A way to survive from the brink of death??" <https://www.binance.com/en/square/post/746243>. 2025-04-15.
- [141] OKLink Academy, "Hot: Inverse finance loses over \$15m in oracle manipulation." https://www.oklink.com/academy/en/2022/07/12/hot-inverse-finance-loses-over-15m-in-oracle-manipulation?utm_source=chatgpt.com. 2025-04-14.
- [142] Blockbasis, "Holograph protocol hack: Mitigating the \$14.4 million exploit." <https://www.blockbasis.com/p/holograph-protocol-hack-mitigating-the-14-4-million-exploit>. 2025-03-16.
- [143] Halborn, "Explained: The holograph hack (june 2024)." <https://www.halborn.com/blog/post/explained-the-holograph-hack-june-2024>. 2025-04-03.
- [144] QuillAudits, "Decoding deus dao \$6.5m exploit." <https://medium.com/p/588bbece61f>. 2025-04-15.
- [145] Cyvers.ai, "Deus finance hack: An in-depth analysis of the \$6.5 million loss." <https://cyvers.ai/blog/deus-finance-hack-an-in-depth-analysis-of-the-6-5-million-loss>. 2025-04-15.
- [146] P. Shukla, "Ionic money \$8.5 million exploit: Hack analysis." <https://www.linkedin.com/pulse/ionic-money-85-million-exploit-hack-analysis-piyush-shukla-7j8if>. 2025-02-09.
- [147] Rekt News, "Ionic money." <https://rekt.news/ionic-money-rekt>. 2025-04-17.
- [148] Rekt News, "Roninnetworkrekii." <https://rekt.news/roninnetwork-rekii>. 2025-04-17.
- [149] Shashank, "Roninbridgehack analysis." <https://blog.solidityscan.com/ronin-bridge-hack-analysis-d8ff64b8fe683>. 2025-04-17.
- [150] LI.FI, "Incident Report - 16th July." <https://li.fi/knowledge-hub/incident-report-16th-july/>. 2025-04-17.
- [151] SolidityScan, "LI.FI Hack Analysis." <https://blog.solidityscan.com/li-fi-hack-analysis-521388128d22>. 2025-04-17.
- [152] SlowMist, "SlowMist Monthly Security Report: Total Losses for July Approx. \$279 Million." <https://medium.com/p-2bb63973d4bf>. 2025-04-17.
- [153] Rekt, "Yearn: Re-rekt." <https://rekt.news/yearn2-rekt>. 2025-04-17.
- [154] CertiK, "Misconfigured earnings: Yearn finance exploit explained." <https://www.certik.com/resources/blog/40AqeRsWi5Hof9etrvjH2-earnings-misconfigured-yearn-finance-exploit-explained>. 2025-04-17.
- [155] SlowMist, "In-depth analysis of zkland hack linked to eralend hack." <https://slowmist.medium.com/in-depth-analysis-of-zkland-hack-linked-to-eralend-hack-fba4af9b66ef>. 2025-04-18.
- [156] P. Shukla, "Zkland hack analysis." <https://www.linkedin.com/pulse/zkland-hack-analysis-piyush-shukla-jsxzf/>. 2025-04-18.
- [157] LI.FI Team, "Incident report – 16th july 2023." <https://li.fi/knowledge-hub/incident-report-16th-july/>. 2025-04-18.
- [158] Rekt News, "Li.fi jumper." <https://rekt.news/lifi-jumper-rekt>. 2025-04-18.
- [159] R. News, "Hundred finance 2." <https://rekt.news/hundred-rekt2>. 2025-04-19.
- [160] Cointelegraph, "Hundred finance hacker moves assets year after \$7m exploit." <https://cointelegraph.com/news/hundred-finance-hacker-moves-assets-year-after-exploit>. 2025-04-19.
- [161] R. News, "Abracadabra." <https://rekt.news/abra-rekt>. 2025-04-19.
- [162] CertiK, "Lodestar finance incident analysis." <https://www.certik.com/resources/blog/TqTyq4vYH18JzS7zyJye9-lodestar-finance-incident-analysis>. 2025-04-18.
- [163] Rekt News, "Lodestar finance." <https://rekt.news/lodestar-rekt>. 2025-04-18.
- [164] REKT, "1inch." <https://rekt.news/1inch-rekt>. 2025-04-18.
- [165] Decurity, "Yul Calldata Corruption: 1inch Postmortem." <https://blog.decurity.io/yul-calldata-corruption-1inch-postmortem-a7ea7a53bfd9>. 2025-04-18.
- [166] BlockApex, "Shezmu hack analysis." <https://medium.com/p/32414e67c7a0>. 2025-04-18.
- [167] CyberStrategy1, "Crypto security truths — issue #13." <https://medium.com/p/098b78916272>. 2025-04-18.
- [168] REKT News, "Gamma strategies." <https://rekt.news/gamma-strategies-rekt>. 2025-04-18.
- [169] Verichains, "Gamma protocol exploit analysis." <https://blog.verichain.sio/p/gamma-protocol-exploit-analysis>. 2025-04-18.
- [170] R. News, "Conic finance." <https://rekt.news/conic-finance-rekt>. 2023-07-10.
- [171] Shashank, "Gamma hack analysis." <https://blog.solidityscan.com/gamma-hack-analysis-6c074e61709e>. 2024-01.
- [172] H. Security, "Explained: The kiloex hack." <https://www.halborn.com/blog/post/explained-the-kiloex-hack-april-2025>. 2025-04-18.
- [173] R. News, "Kiloex." <https://rekt.news/kiloex-rekt>. 2025-04-18.
- [174] Cointelegraph, "Sir.trading begs hacker to return \$255k or 'no chance for us to survive'." <https://www.tradingview.com/news/cointelegraph-6a2330d53094b>. 2025-04-18.
- [175] R. News, "Sir.trading." <https://rekt.news/sirtrading-rekt>. 2025-04-18.
- [176] H. Security, "Explained: The abracadabra money hack (march 2025)." <https://www.halborn.com/blog/post/explained-the-abracadabra-money-hack-march-2025>. 2025-04-22.
- [177] REKT, "DeltaPrime II." <https://rekt.news/deltaprime-rekt2>. 2025-04-22.
- [178] SolidityScan, "Deltaprime hack analysis." <https://blog.solidityscan.com/m/deltaprime-hack-analysis-44edb9b22567>. 2025-04-22.

- [179] Halborn, “Explained: The onyx protocol hack.” https://www.halborn.com/blog/post/explained-the-onyx-protocol-hack-september-2024?utm_source=chatgpt.com. 2024-04-23.
- [180] REKT, “Onyx protocol ii.” <https://rekt.news/onyx-protocol-rekt2>. 2024-04-23.
- [181] Rekt News, “Dexible.” <https://rekt.news/dexible-rekt>. 2025-04-23.
- [182] Shashank, “Dexible hack analysis.” <https://blog.solidityscan.com/dexible-hack-analysis-never-blindly-trust-smart-contracts-c2aee7943c1a>. 2025-04-23.
- [183] D. Security, “Dough finance incident analysis.” <https://medium.com/p/04db15c22552>. 2025-04-22.
- [184] CertiK, “Dough finance incident analysis.” <https://www.certik.com/resources/blog/3SMOUGMCSStY4pQW6I49W2-dough-finance-incident-analysis>. 2025-04-22.
- [185] SolidityScan Team, “Clobberdex liquidity vault hack analysis.” <https://blog.solidityscan.com/clobberdex-liquidity-vault-hack-analysis-f22eb960aa6f>. 2025-04-25.
- [186] REKT News, “Clobberdex.” <https://rekt.news/clobberdex-rekt>. 2025-04-25.
- [187] Rekt News, “Tornado cash governance attack.” <https://rekt.news/tornado-gov-rekt>. 2025-04-25.
- [188] R. Behnke, “Explained: The tornado cash hack (may 2023).” <https://www.halborn.com/blog/post/explained-the-tornado-cash-hack-may-2023>. 2025-04-25.
- [189] REKT, “Team finance.” <https://rekt.news/teamfinance-rekt>.
- [190] SlowMist, “Analysis & review of team finance exploit.” <https://medium.com/p/f439c2f63e2>. 2025-06-12.
- [191] Rekt News, “Raft rekt.” <https://rekt.news/raft-rekt>. 2025-04-25.
- [192] SharkTeam, “Sharkteam analysis of the principles of raft attack incident.” <https://medium.com/p/814127d626bd>. 2025-04-25.
- [193] REKT, “Rho market.” <https://rekt.news/rho-market-rekt>. 2025-04-28.
- [194] Olympix, “Rho markets on scroll exploit analysis.” <https://medium.com/p/965991270f56>. 2025-04-28.
- [195] REKT, “Uwulend.” <https://rekt.news/uwulend-rekt>. 2025-04-28.
- [196] Shashank, “Uwulend hack analysis.” <https://blog.solidityscan.com/uwulend-hack-analysis-77eb9181a717>. 2025-04-28.
- [197] REKT, “Velocore.” <https://rekt.news/velocore-rekt>. 2025-04-28.
- [198] VELOCORE, “Velocore incident postmortem.” <https://medium.com/p/6197020ec3e9>. 2025-04-28.
- [199] REKT, “Curio.” <https://rekt.news/curio-rekt>. 2025-04-28.
- [200] R. Behnke, “Explained: The curio hack (march 2024).” <https://www.halborn.com/blog/post/explained-the-curio-hack-march-2024>. 2025-04-28.
- [201] R. News, “Unizen rekt.” <https://rekt.news/unizen-rekt>. 2025-04-29.
- [202] Blockbasis, “Unizen: Navigating the \$2.1m security breach.” <https://blockbasis.com/p/unizen-navigating-21m-security-breach>. 2025-04-29.
- [203] Rekt, “Seneca protocol.” <https://rekt.news/seneca-protocol-rekt>. 2025-04-30.
- [204] BlockApex, “Seneca protocol hack analysis.” <https://medium.com/p/46f3bcc1040>. 2025-04-30.
- [205] Rekt News, “OKX DEX.” <https://rekt.news/okx-dex-rekt>. 2025-04-30.
- [206] O. AI, “Unmasking the malicious okx dex upgrade.” <https://medium.com/p/f6912449f963>. 2025-04-30.
- [207] REKT News, “Zunami protocol.” <https://rekt.news/zunami-protocol-rekt>. 2025-04-30.
- [208] SolidityScan, “Zunami protocol hack analysis.” <https://blog.solidityscan.com/zunami-protocol-hack-analysis-e95981976e11>. 2025-04-30.
- [209] Rekt News, “EraLend REKT: Read-only Re-entrancy Exploit Steals \$3.4M.” <https://rekt.news/eralend-rekt>. 2025-04-29.
- [210] Bitcoinist, “Eralend exploit.” <https://cryptorank.io/news/feed/ad957-era-lend-exploit-hackers-steal-3-4-million-dollars>. 2025-04-29.
- [211] Rekt News, “Atlantis loans.” <https://rekt.news/atlantis-loans-rekt>. 2025-04-29.
- [212] Shashank, “Atlantis loans hack analysis.” <https://blog.solidityscan.com/atlantis-loans-hack-analysis-7f3fb2e295e0>. 2025-04-29.
- [213] R. News, “Sturdy.” <https://rekt.news/sturdy-rekt>. 2025-05-04.
- [214] S. Team, “Sturdy finance hack analysis.” <https://blog.solidityscan.com/sturdy-finance-hack-analysis-bd8605cd2956>. 2025-05-04.
- [215] Rekt, “Jimbos protocol \$7.5 million hack.” <https://rekt.news/jimbo-rekt>. 2023-05-29.
- [216] N. C. Labs, “A detailed analysis of arbitrum-based jimbos protocol’s \$7.5 million hack.” <https://medium.com/p/36af84face2>. 2023-05-29.
- [217] Rekt News, “Swaprum rekt.” <https://rekt.news/swaprum-rekt>. 2024-04-20.
- [218] CryptoRank, “The swaprum incident.” <https://cryptorank.io/ru/news/fe/ed/cdce4-the-swaprum-incident-audited-defi-protocol-dupes-investors-out-3m>. 2025-04-17.
- [219] Rekt News, “Orion protocol.” <https://rekt.news/orion-protocol-rekt>. 2025-05-04.
- [220] SlowMist, “An analysis of the attack on orion protocol.” <https://medium.com/p/c7aef70aff83>. 2025-05-04.
- [221] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
- [222] Compound, “Compound governance examples.” <https://github.com/compound-developers/compound-governance-examples>. 2025-01-16.
- [223] J. Ellul and G. J. Pace, “SoliNomic: A Self-modifying Smart Contract Game Exploring Reflexivity in Law,” in *Disruptive Technologies in Media, Arts and Design* (A. Dingli, A. Pfeiffer, A. Serada, M. Bugeja, and S. Bezzina, eds.), (Cham), pp. 3–14, Springer International Publishing, 2022.
- [224] S. Team, “Solidity v0.5.0 breaking changes.” <https://docs.soliditylang.org/en/latest/050-breaking-changes.html>. 2025-02-10.
- [225] V. Piantadosi, G. Rosa, D. Placella, S. Scalabrino, and R. Oliveto, “Detecting functional and security-related issues in smart contracts: A systematic literature review,” *Software: Practice and experience*, vol. 53, no. 2, pp. 465–495, 2023.
- [226] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, “Securify: Practical security analysis of smart contracts,” in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 67–82, 2018.
- [227] Ethereum Foundation, “Solidity v0.5.0 breaking changes.” <https://docs.soliditylang.org/en/v0.5.0/050-breaking-changes.html>. 2025-02-22.
- [228] E. Foundation, “EIP-6780: SELFDESTRUCT only in same transaction as creation.” <https://eips.ethereum.org/EIPS/eip-6780>. 2025-03-22.
- [229] M. Eshghie, C. Artho, H. Stammel, W. Ahrendt, T. Hildebrandt, and G. Schneider, “Highguard: Cross-chain business logic monitoring of smart contracts,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE ’24*, p. 2378–2381, ACM, 2024.
- [230] P. Tolmach, Y. Li, S.-W. Lin, and Y. Liu, “Formal analysis of composable DeFi protocols,” in *Financial Cryptography and Data Security. FC 2021 International Workshops* (M. Bernhard, A. Bracciali, L. Gudgeon, T. Haines, A. Klages-Mundt, S. Matsuo, D. Perez, M. Sala, and S. Werner, eds.), (Berlin, Heidelberg), pp. 149–161, Springer Berlin Heidelberg, 2021.
- [231] E. I. Proposals, “EIP-1153: Transient storage opcodes.” <https://eips.ethereum.org/EIPS/eip-1153>. 2025-06-28.
- [232] Foundry Team, “Foundry-rs/foundry.” <https://github.com/Foundry-rs/Foundry>. 2025-07-21.
- [233] Rekt News, “Inverse finance.” <https://rekt.news/inverse-finance-rekt>. 2023-05-15.
- [234] J. Fei, X. Chen, and X. Zhao, “Msmart: Smart contract vulnerability analysis and improved strategies based on smartcheck,” *Applied Sciences*, vol. 13, no. 3, p. 1733, 2023.