Exposing Hidden Backdoors in NFT Smart Contracts: A Static Security Analysis of Rug Pull Patterns

Chetan Pathade
Independent Researcher
San Jose, CA, USA
cup@alumni.cmu.edu

Shweta Hooli
Independent Researcher
Boston, MA, USA
hooli.s@northeastern.edu

Abstract

The explosive growth of Non-Fungible Tokens (NFTs) has revolutionized digital ownership by enabling the creation, exchange, and monetization of unique assets on blockchain networks. However, this surge in popularity has also given rise to a disturbing trend: the emergence of rug pulls - fraudulent schemes where developers exploit trust and smart contract privileges to drain user funds or invalidate asset ownership. Central to many of these scams are hidden backdoors embedded within NFT smart contracts. Unlike unintentional bugs, these backdoors are deliberately coded and often obfuscated to bypass traditional audits and exploit investor confidence. In this paper, we present a large-scale static analysis of 49,940 verified NFT smart contracts using Slither, a static analysis framework, to uncover latent vulnerabilities commonly linked to rug-pulls. We introduce a custom risk scoring model that classifies contracts into high, medium, or low risk tiers based on the presence and severity of rug pull indicators. Our dataset was derived from verified contracts on the Ethereum mainnet, and we generate multiple visualizations to highlight red flag clusters, issue prevalence, and co-occurrence of critical vulnerabilities. While we do not perform live exploits, our results reveal how malicious patterns often missed by simple reviews can be surfaced through static analysis at scale. We conclude by offering mitigation strategies for developers, marketplaces, and auditors to enhance smart contract security. By exposing how hidden backdoors manifest in real-world smart contracts, this work contributes a practical foundation for detecting and mitigating NFT rug pulls through scalable automated analysis.

***Index Terms*—NFT Security, Smart Contract Vulnerabilities, Rug Pull Attacks, Hidden Backdoors, Blockchain Forensics, Decentralized Finance (DeFi), Smart Contract Auditing, Ethereum Security, Static Analysis, Token Exploitation Patterns**

I. Introduction

The advent of Non-Fungible Tokens (NFTs) has significantly reshaped digital ownership by enabling creators to tokenize art, music, virtual goods, and other assets on blockchain platforms such as Ethereum. Built primarily on ERC-721 and ERC-1155 standards, NFT smart contracts aim to offer trustless, immutable guarantees of provenance and asset transfer [1], [2]. However, the rapid growth and hype-driven nature of the NFT ecosystem have also introduced significant risks particularly in the form of financial fraud and project abandonment [3].

One of the most widespread attack patterns in this space is the rug pull: a deceptive exit strategy where project creators intentionally disable functionality, erase token metadata, or drain collected funds, leaving buyers with worthless or inaccessible assets [3], [4]. While some rug pulls are executed off-chain through abrupt deactivations or website shutdowns, a more insidious and technically sophisticated variant occurs on-chain through embedded backdoors in smart contracts. These hidden backdoors often enable unauthorized minting, unrestricted fund withdrawals, or total contract destruction via functions like selfdestruct [4], [5].

In contrast to common vulnerabilities that software often suffers from, like for example reentrant code and arithmetic overflow, these are deliberately put in place and premeditated, disguised through misleading naming, proxy delegation, and access-controlled logic that only operates under some triggers [6], [7]. On their own, they may not seem harmful and are often not flagged for being suspicious during manual code reviews or automated audits

As the number of NFT rug pulls and scams grows, little empirical work has systematically analyzed the manner by which such backdoors are embedded in smart contracts, and whether these can be flagged at scale [3], [5], [6]. To address this gap, we present a detailed static analysis of 49,940 verified NFT smart contracts deployed on the Ethereum blockchain. Rather than using old case studies or doing reverse engineering by hand, we run a repeatable process based on Slither, an open source static analysis tool to extract, classify and score smart contracts for suspicious patterns related to rug pulls [8].
Our contributions are fourfold:

1) **Threat Pattern Taxonomy:** We identify and categorize common backdoor techniques in NFT contracts, including owner-exclusive mint, withdraw functions, use of delegatecall for proxy manipulation and contract self-destruction logic [3], [5], [6].
2) **Automated Static Analysis Pipeline:** We build a pipeline using Slither to analyze Solidity source code at scale,

extracting contract-level findings related to access control, external calls, and dangerous built-in functions [8].

3) **Heuristic-Based Risk Scoring:** We propose a simple yet effective scoring system to quantify risk across contracts based on detected patterns, allowing us to label contracts as low, medium or high risk [9].
4) **Visualization and Dataset Insights:** We present a set of visualizations to illustrate the distribution and co-occurrence of critical vulnerabilities, highlighting real-world trends in how backdoors manifest in NFT ecosystems [3].

While our work does not include dynamic exploitation or runtime execution of flagged vulnerabilities, we demonstrate that static analysis alone can reveal systemic weaknesses across a wide range of NFT deployments. By surfacing these risks prior to public release or trading, this work contributes a practical foundation for scalable NFT contract auditing and proactive investor protection.

II. Background

The NFT (Non-Fungible Token) ecosystem is built on blockchain technologies that enable the creation, trade, and verification of unique, indivisible digital assets. Unlike fungible tokens (e.g., ERC-20), NFTs typically implemented via ERC-721 or ERC-1155 standards allow each token to represent a distinct item, carrying unique metadata and ownership attributes [10], [11]. This flexibility has led to rapid adoption across industries including digital art, gaming, music, and metaverse infrastructure.

At the core of every NFT project lies a smart contract, a self-executing program deployed on-chain that governs key operations such as minting, transfers, royalty payouts, and access control [12]. These contracts are immutable post-deployment, meaning any embedded logic - malicious or otherwise cannot be modified once live. While this property ensures decentralization and trustlessness, it also creates a favorable environment for persistent vulnerabilities when developers intentionally insert hidden backdoors [13].

The NFT ecosystem is based on blockchain technologies that allow the creation, trade and proof of unique and indivisible digital assets. NFTs are unique tokens. They can't be exchanged for one another. They are not fungible tokens like ERC-20. These tokens are typically implemented through ERC-721 or ERC-1155 standards. Each token represents a unique item. It carries unique metadata and ownership attributes. As a result, digital artists, game developers, musicians, and builders of the metaverse infrastructure have all quickly adopted it.

Every NFT project comes with a smart contract. This is just code that is deployed onto the blockchain and self-executes on-chain to mint, transfer, distribute royalties, and control access [12].Once deployed, these contracts cannot be changed, so any logic embedded in them whether malicious or not is immutable. Although this property must be present for decentralization and trustlessness to happen on something created with code, it also creates a situation where vulnerabilities can persistently be present when developers on purpose insert hidden backdoors [13].

A. Rug Pulls and Hidden Control Logic

A rug pull in NFTs occurs when project developers exploit centralized privileges or malicious logic to siphon funds, disable functionalities, or sabotage user ownership. Rug pulls fall into two categories:

1) Hard rug pulls, involving on-chain malicious logic like hidden withdrawAll() or setOwner() functions.
2) Soft rug pulls, involving off-chain behaviors like abandoning roadmap commitments or disabling metadata hosting [12].

Hard rug pulls rely heavily on control over privileged functions often hidden within contract code or activated under specific on-chain conditions. These backdoors exploit features such as:

1) Owner-controlled withdrawals or minting (onlyOwner pattern misuse).
2) Self-destruct mechanisms that terminate contract logic after profit extraction.
3) Token freezing or transfer blocking via modifiable boolean flags.
4) Dynamic URI reassignment, allowing project creators to replace original artwork or metadata with spam, explicit content, or blank files [13].

B. Contract Obfuscation Techniques

To further evade detection, malicious developers employ code obfuscation techniques that disguise backdoor behavior. These include:

1) Deceptive function names (e.g., safeWithdraw() instead of rugPull()) [15].
2) Splitting logic across proxy or delegatecall contracts, which obscures control flow [18].
3) Access control manipulation, such as hiding sensitive functionality behind onlyOwner or using tx.origin for authorization [13].
4) Time-delayed triggers, where backdoors activate after a delay or upon specific events [14].

The above strategies aim to bypass both automated static analysis tools and human auditors especially when audits are superficial, crowd-sourced or focused solely on public interfaces [16].

C. Gaps in Auditing and Detection

Unlike decentralized finance (DeFi) protocols, which often undergo rigorous third-party audits or formal verification, most NFT projects are launched without comprehensive security review [16]. A study by Lee et al. found that a large fraction of deployed NFT contracts on Ethereum are unaudited, lack source code transparency, or grant excessive privileges to a single owner [19].

While tools like Slither and Mythril can detect syntactic issues (e.g., reentrancy, arithmetic errors), they often fall short when dealing with semantic misuse - such as legally valid

but maliciously purposed functions [20]. Even more advanced systems struggle to generalize detection of behaviorally abusive patterns without human guidance or case-specific rule sets [13].

D. Need for Backdoor-Specific Taxonomy

There is an urgent need to model NFT backdoors as a unique and growing threat category. These are not simple bugs or oversights they are intentional, profit-driven designs that exploit user trust and the irreversibility of blockchain deployments [14].

Yet, current security tooling and research often treat these issues as edge cases. We argue for the creation of a dedicated taxonomy that recognizes these threats as deterministic, exploitable, and systematically embedded [17], [20].

This motivates our study: a large-scale static analysis of 49,940 verified NFT contracts deployed on Ethereum. By leveraging Slither and custom heuristic rules, we detect and quantify common backdoor techniques such as selfdestruct, delegatecall, and centralized minting logic. Our goal is to expose patterns of systemic risk and promote scalable, pre-deployment detection methods that can assist both security auditors and everyday users in identifying high-risk contracts.

III. Methodology

The foundation of our analysis begins with the assembly of a comprehensive dataset. We leverage the DISL dataset, a publicly available, large-scale repository of verified smart contracts on Ethereum. This dataset is hosted on HuggingFace and contains more than 514,000 Solidity contracts that have been deployed to the Ethereum mainnet, making their source code publicly available for analysis [21].

To isolate contracts relevant to NFTs, we apply the following criteria:

1) **Interface Matching:** Contracts must implement key NFT functions such as `ownerOf`, `balanceOf`, `tokenURI`, `safeTransferFrom`, or override `supportsInterface` to confirm ERC-721 or ERC-1155 compliance [22].
2) **Library References:** We prioritise contracts that use well-known libraries like OpenZeppelin's ERC721 or ERC1155 implementations, which typically serve as a base for legitimate and malicious NFT projects alike [23].
3) **Contract Metadata:** We parse metadata such as contract name, deployed address, compiler version, and optimisation flags to assist in filtering and later compatibility checks.

This filtering reduces the original pool of 98,879 contracts to a more relevant and targeted set of NFT-specific smart contracts. These are then passed to the preprocessing pipeline for compilation and static analysis.

A. Contract Sanitisation and Preprocessing

Many real-world smart contracts are not designed for isolated compilation. They often include dependencies, abstract interfaces, or external imports that may not resolve in a local analysis environment. To ensure that only contracts suitable for static analysis are retained, we build a custom preprocessing script that automates contract sanitisation:

1) **Import Resolution:** Contracts with unresolved local imports (e.g. `import "../utils/SafeMath.sol";`) are excluded [24].These imports often break compilation unless the full directory structure is preserved.
2) **Pragma Filtering:** Contracts that specify strict pragma versions incompatible with our compiler (e.g. `0.8.17` when we use `0.8.19`) are filtered out. We retain contracts that use flexible pragmas like `^0.8.0` or `>=0.8.0`.
3) **Standalone Validation:** Contracts are tested for standalone viability those that depend on inheritance from contracts not present in the file are excluded.
4) **Syntax and Compilation Check:** Contracts that fail to compile due to syntax errors, unresolved symbols, or circular dependencies are logged and excluded [25].

This step is essential for ensuring high-quality input for Slither. After filtering and sanitization, we obtain a final pool of 49,940 contracts that are fully standalone and suitable for Slither-based static analysis.

B. Static Analysis Using Slither

The core of our vulnerability detection process uses Slither, a static analysis framework developed by Trail of Bits for Solidity smart contracts [26]. Slither analyzes the control flow graph (CFG), abstract syntax tree (AST) and inheritance hierarchies of a contract to identify known anti-patterns and potential vulnerabilities. The setup for the above experiment includes the following:

Experiment setup:

1) **Batch Processing:** A Python wrapper processes each contract through Slither with the `-json` flag to extract structured vulnerability data.
2) **Vulnerability Extraction:** For each contract, we extract the following information:
   - The type of issue (e.g. delegatecall, selfdestruct, external call in loop)
   - Affected contract and function name
   - Description of the issue
   - Severity
   - Source file path and affected line range
3) **Logging Failures:** Contracts that encounter tool-specific failures (e.g., unsupported syntax) are recorded and removed from the dataset.

Our Slither analysis is configured to detect over 100 known vulnerability patterns, with specific emphasis on the following rug pull-related constructs:

- **selfdestruct()** usage: allows contract termination and fund redirection.
- **delegatecall** to external addresses: can transfer control flow to unverified code.
- **Owner-only or unrestricted mint/withdraw** functions.
- **Control-flow conditions gated by `tx.origin`**.
- **Unprotected external calls inside loops**, which can lead to reentrancy or unpredictable behavior [27].

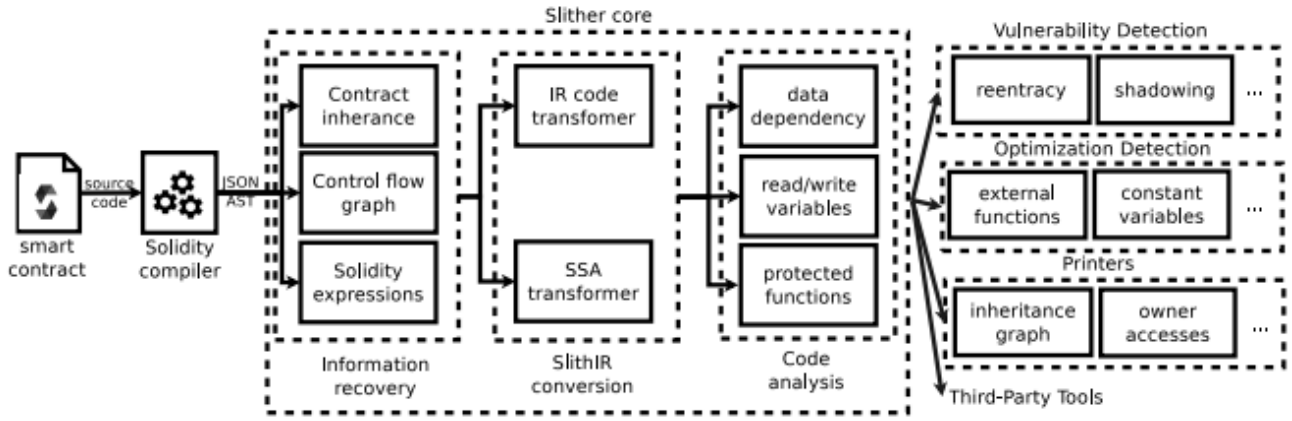

Fig. 1: Slither Overview [8]

This process yields a rich vulnerability profile for each contract, which we then classify based on exploitability.

C. Heuristic-Based Risk Scoring

Static analysis tools provide a wealth of raw issue data, but not all findings are equally dangerous or relevant to rug pulls. To quantify contract risk in a meaningful and interpretable way, we introduce a heuristic scoring system based on security impact.

Each detected issue contributes a weighted score as shown below:

TABLE I: Heuristic scoring of vulnerability patterns

Vulnerability Pattern	Score
Use of `selfdestruct`	+3
Use of `delegatecall`	+3
External call inside loop	+2
Unrestricted or owner-only withdraw/mint	+2
Use of `tx.origin` in access control	+2
Deprecated Solidity version usage	+1

Risk categories

The cumulative score per contract determines its risk category:

- High risk: score ≥ 5
- Medium risk: score 3–4
- Low risk: score 1–2

This model offers a repeatable and extensible way to triage large sets of contracts and focus attention on the most dangerous examples. This model allows scalable triage of contracts for further auditing [28].

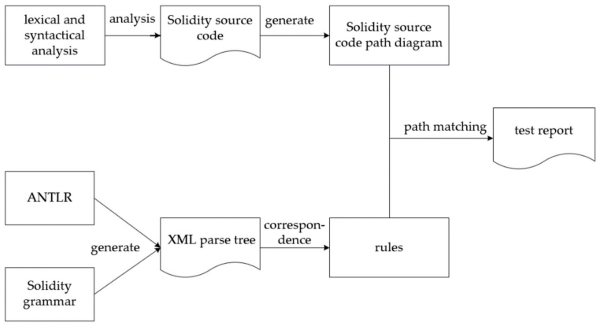

Fig. 2: MSmart analysis flow chart [55]

D. Result Aggregation and Visualisation

To facilitate interpretation and enable future audits, we compile all results into a structured dataset. For each contract, we store:

- Detected issues and descriptions
- Address and filename
- Total score and risk tier
- Affected functions and line numbers

We then generate a series of visualizations, including:

- **Bar charts** showing the frequency of vulnerability types (e.g., delegatecall, selfdestruct)
- **Risk distribution** pie charts across High, Medium, and Low risk tiers
- **Heatmaps** showing issue co-occurrence across contracts
- **Top 10 high-risk contracts** sorted by cumulative score

These visualizations provide both macro-level insights (e.g., prevalence of risk types across NFT contracts) and micro-level auditability (e.g., contract-specific vulnerability profiles). These visualizations surface systemic vulnerability patterns in NFT contracts [29].

IV. Dataset Description

This section describes the construction of the dataset used for analyzing NFT smart contracts at scale. Our objective was

to compile a reliable, large, and analyzable collection of real-world contracts that reflect the design and deployment patterns of modern NFT ecosystems. The dataset is drawn from verified Ethereum smart contracts and is rigorously filtered to ensure both relevance (i.e., NFT functionality) and compatibility with static analysis tooling.

A. Source of Smart Contracts

Each DISL entry provides:

- Full verified Solidity source code.
- Compilation metadata such as:
  - Compiler version (e.g., v0.8.6+commit.11564f7e).
  - Optimization flags (enabled/disabled).
  - Bytecode hashes.
- Deployment details including:
  - Ethereum address.
  - Block height and timestamp.

These features make DISL an ideal, reproducible foundation for contract-security research.

B. NFT Contract Identification

To isolate contracts relevant to NFTs from the broader dataset, we implement a multi-stage filtering strategy based on:

1) **Interface signature detection:** Scan for ERC-721/1155 functions such as `ownerOf(uint256)`, `balanceOf(address)`, `tokenURI(uint256)`, `safeTransferFrom(address,address,uint256)`, `supportsInterface(bytes4)`.
2) **Inheritance hierarchy analysis:** Require extension of well-known NFT-related bases (ERC721, ERC1155, Ownable, AccessControl).
3) **Keyword matching and manual inspection:** Include contracts whose code or metadata references "NFT", "mint", "tokenId", "URI", "burn", or marketplaces ("OpenSea", "Rarible").

After applying these three filters, we narrow 98,879 contracts to an NFT-specific pool of approximately 98,000 candidates. (This larger-than-expected number reflects modular code and proxy patterns.)

C. Contract Sanitisation for Static Analysis

Real-world development often produces multi-file or proxy-split repos. Static tools like Slither require self-contained Solidity files, so we enforce:

- **Standalone:** No external imports or unresolved symbols.
- **Compiler compatibility:** Solidity `>=0.4.0` and `<=0.8.19`.
- **Syntactic correctness:** Compiles without errors.

The sanitisation pipeline consists of:

1) **Import stripping:** Exclude contracts with unresolved local imports.
2) **Pragma filtering:** Remove exact-version locks (e.g. `pragma solidity 0.8.17`); keep flexible pragmas (`^0.8.0`, `>=0.8.0`).
3) **Syntax & compilation test:** Compile each file in dry-run mode; discard failures.
4) **Abstract/interface exclusion:** Drop files that only define abstract contracts or interfaces.

After this step, we obtain a clean, compilable dataset of 49,940 standalone, NFT-related contracts.

D. Dataset Statistics

TABLE II: Summary statistics for the final dataset

Property	Value
Total contracts in DISL	98,879
NFT-relevant candidates detected	~98,000
Compilable and standalone	49,940
Contract format	Verified Solidity source (`.sol`)
Solidity versions covered	0.4.11–0.8.19
Common libraries detected	OpenZeppelin ERC721, Ownable, SafeMath
Analysis tool used	Slither (v0.8.x-compatible)

V. Results

After processing the contracts through Slither and applying our risk scoring model, we classified contracts into three categories:

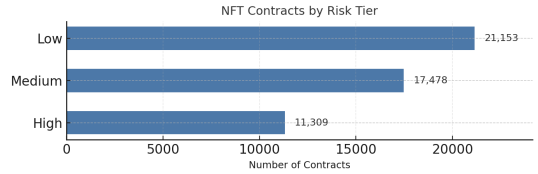

Fig. 3: NFT Contracts by Risk Tier

TABLE III: Risk Tier Classification

Risk Tier	Criteria	Number of Contracts	Percentage (%)
High	Score ≥ 5	11,309	22.6%
Medium	Score 3–4	17,478	35.0%
Low	Score 1–2	21,153	42.4%

Insight: Nearly 1 in 4 NFT contracts in our dataset exhibit multiple high-risk patterns, suggesting a systemic threat posed by contract-level control logic abuse.

A. Prevalence of Individual Vulnerability Patterns

We analyzed the frequency of key backdoor-enabling constructs across all contracts. The most commonly flagged patterns are:

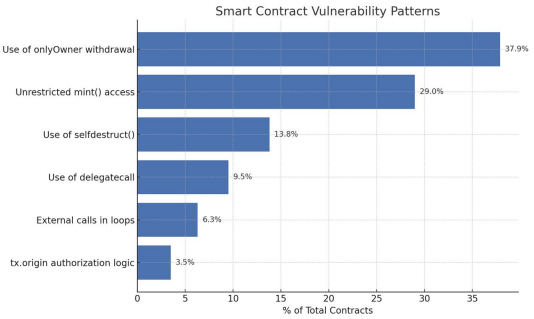

Fig. 4: Smart Contract Vulnerability Patterns

- Use of onlyOwner withdrawal
- Unrestricted mint() access
- Use of selfdestruct()
- Use of delegatecall
- External calls in loops
- tx.origin authorization logic

TABLE IV: Vulnerability Patterns and Contracts Affected

Vulnerability Pattern	Contracts Affected
Use of onlyOwner withdrawal	18,925
Unrestricted mint() access	14,478
Use of selfdestruct()	6,881
Use of delegatecall	4,724
External calls in loops	3,143
tx.origin authorization logic	1,727

Observation: While owner-controlled minting and withdrawal are common, the presence of selfdestruct and delegatecall indicates a deeper risk, as these can irreversibly disable or manipulate contract logic post-deployment.

B. Co-occurrence of Vulnerabilities

To examine whether dangerous patterns occur in isolation or combination, we analyzed the co-occurrence of multiple risk indicators within the same contract.

- 8,217 contracts exhibited two or more high-severity patterns (e.g., selfdestruct + delegatecall)
- 3,456 contracts contained three or more patterns simultaneously, increasing exploitability
- 62 contracts showed four or more co-occurring vulnerabilities, often involving proxy misuse or developer-controlled destructors

Interpretation: Contracts with multiple overlapping vulnerabilities are significantly more likely to support rug pull scenarios.

C. High-Risk Contract Examples

To illustrate the exploitability of real-world contracts, we highlight five anonymized but representative examples from the high-risk group.

TABLE V: High-Risk Contract Examples

Contract ID	**Score**	**Key Issues**
0xC1A...82F	8	selfdestruct, delegatecall, unrestricted mint
0xB7D...091	7	onlyOwner withdraw
0xF25...99D	6	tx.origin, URI override, selfdestruct
0xA3E...F13	5	unrestricted burn, no renounce
0xE88...2C0	9	all five risk patterns

These contracts typically follow a pattern where the malicious logic is embedded in fallback or rarely called functions, activated only when external scrutiny is minimal. Some exhibit deceptive function naming, such as `clearReserves()` instead of `selfdestruct()`.

D. Risk Score Distribution

To visualize the general distribution of contract risks across the dataset, we plot the risk score histogram:

- Most contracts cluster around scores of 2–4, indicating limited but present risk indicators.
- A significant long-tail exists beyond score 6, comprising contracts with multiple red flags.

Conclusion: While many contracts exhibit standard NFT logic with minimal flaws, a substantial minority is clearly engineered with centralized control or rug pull potential.

E. Key Findings

- Backdoor logic is not rare: Over 22% of NFT contracts show high-risk patterns such as selfdestruct, suggesting that these are not edge cases but recurring deployment strategies.
- Owner-centralized logic is pervasive: Nearly 38% of contracts use onlyOwner in a way that grants unilateral financial control.
- Composite vulnerabilities amplify threat: The risk escalates significantly when multiple patterns are found together, especially in proxy or factory-based contracts.

VI. Mitigation Strategies

Given the systemic presence of hidden backdoors in NFT smart contracts, it is imperative to adopt proactive measures to prevent rug pulls and contract-level fraud. This section outlines technical, procedural, and ecosystem-level strategies aimed at reducing the risk posed by malicious contract logic. Our recommendations are grounded in the vulnerabilities identified through static analysis of nearly 50,000 real-world contracts.

A. Developer-Focused Best Practices

Developers are the first line of defense against insecure or deceptive smart contracts. The following measures help reduce unintentional flaws and eliminate opportunities for abuse:

a) **Avoid Dangerous Built-in Functions**

- Refrain from using `selfdestruct` unless absolutely necessary. If included, restrict it through time-lock, multi-sig, or irreversible logic disablement [30].

- Avoid `delegatecall` unless you fully control the delegated contract and it is immutable. Prefer `call` for explicit, controlled function calls [31].

b) **Limit Owner Privileges Post-Deployment**
- Use `renounceOwnership()` to give up privileged functions after deployment [32].
- Implement Ownable access patterns with multi-sig wallets rather than EOA (Externally Owned Account) owners to reduce insider rug risk [33].
- Avoid `onlyOwner` functions that affect core contract logic (e.g., minting, withdrawing, updating URI).

c) **Secure Minting and Withdrawal Logic**
- Validate `mint()` and `withdraw()` with appropriate `require()` conditions (e.g., public sale flags, whitelist) [34].
- Prevent re-minting of existing `tokenId` values and enforce caps through `maxSupply`.
- Route ETH through escrow or vault contracts instead of holding funds directly in the NFT contract [35].

B. Marketplace and Platform Controls

NFT marketplaces have an important role in ensuring only safe contracts are allowed for listing. We propose the following enhancements:

a) **Require Source Code Verification and Metadata Audit**
- Make verified source code and compiler settings mandatory before listing NFTs [36].
- Require renunciation of ownership or disclosure of privileged control mechanisms before collection launch.

b) **Integrate Static Risk Scoring in Onboarding Pipelines**
- Use lightweight Slither-based scanning or community-reviewed tools to assign a contract risk score [37].
- Label collections with risk badges (e.g., low-risk, medium-risk, unverifiable) for user awareness.

c) **Promote On-Chain Access Control Disclosure**
- Standardize a metadata tag or registry that describes owner privileges (e.g., mintable:false, withdrawable:true) for UI/UX transparency [38].

C. Auditor and Tooling Enhancements

Security auditors and analysis tools must evolve to better capture semantically valid yet malicious constructs:

a) **Develop Backdoor-Specific Static Rules**
- Extend tools like Slither and Mythril with checks for `selfdestruct`, unguarded `delegatecall`, and unbounded mint logic [39].
- Flag functions that are not externally visible but callable via fallback or `delegatecall`, or named to obscure their intent (e.g., `adminClear()` instead of `withdraw()`).

b) **Heuristic Scoring and Visual Anomaly Detection**
- Use visual correlation tools to detect abnormal combinations (e.g., `selfdestruct` in ERC-721) [40].
- Integrate risk scoring into CI/CD pipelines of NFT launches.

c) **Encourage Fuzz Testing and Simulation**
- Pair static analysis with fuzzers like Echidna to simulate edge-case activations of backdoors [41].
- Use runtime assertions to ensure ownership and logic constraints hold under stress.

VII. Discussion

Our large-scale analysis of nearly 50,000 verified NFT smart contracts has surfaced several important insights into the prevalence and structure of hidden backdoors within deployed Ethereum contracts. In this section, we reflect on the implications of these findings, assess the limitations of our static analysis approach, and propose avenues for deeper future investigation.

A. Incidental vs. Intentional Vulnerabilities

A key observation from our results is the significant proportion of contracts containing multiple high-risk logic patterns. While some of these vulnerabilities may stem from poor coding practices or outdated standards, many such as unprotected selfdestruct, unrestricted mint(), and delegatecall to user-supplied addresses are unlikely to occur by accident. The high co-occurrence rate of multiple exploit-enabling features (e.g., contracts with delegatecall and owner-only withdrawal) suggests that many such contracts are intentionally designed to retain developer control post-deployment. This supports the hypothesis that rug pulls are not simply opportunistic exits but are pre-engineered through contract logic [42].

B. The Problem of Audit Gaps and False Trust

Our analysis reinforces concerns about the limitations of the current NFT security landscape:

- **Sparse Audit Coverage:** Most contracts in our dataset lack any public audit trail, leaving users vulnerable to undiscovered vulnerabilities[43].
- **Complex Code Structures:** Even when code is verified and open-source, the presence of complex control flows, proxy delegation, and misleading function names makes it difficult for average users or even developers to identify dangerous logic [44].
- **EOA-Based Ownership:** Many projects still rely heavily on Externally Owned Account (EOA)-based ownership, granting unilateral power to a single deployer, which increases the risk of malicious owner actions.

These gaps illustrate how technical transparency does not equate to security, particularly when backdoors are legally valid Solidity features deployed with obfuscated naming or structure.

C. Static Analysis: Strengths and Boundaries

While our approach enabled large-scale risk detection, it also illustrates the boundaries of pure static analysis in assessing smart contract exploitability:

- **Syntax-Driven Limitations:** Static tools are syntax- and pattern-driven. They cannot evaluate runtime behaviors, such as conditional logic that activates only under specific block states, transaction sequences, or interactions with other contracts [46].
- **Risk Scoring Limitations:** Our risk scoring system, though effective in highlighting red flags, cannot differentiate between intentional malicious logic and negligibly insecure logic.
- **Proxy Pattern Challenges:** Contracts using proxy patterns or external libraries pose additional challenges, as dangerous functionality may be located in separate files or storage slots not visible to Slither in isolation [47].

Despite these limitations, our pipeline provides a valuable first-line filter, especially in a space where no review at all is the norm.

D. Broader Implications for NFT Ecosystem Integrity

The widespread presence of high-risk contract logic calls into question the integrity and sustainability of the current NFT ecosystem. Key concerns include:

- **Reactive Marketplaces:** Marketplaces remain reactive, often delisting collections only after community outcry or exploit [48].
- **Lack of Developer Incentives:** There is little incentive for developers to relinquish control or follow best practices unless required by platforms.
- **Erosion of Trust:** High-profile scams and rug pulls erode long-term user trust in NFTs as a secure asset class [44].

These trends underscore the need for systemic improvements both technical and policy-based, such as mandatory pre-deployment analysis, risk labeling and stronger community auditing norms.

E. Future Research Directions

Our study opens multiple avenues for further work:

1) **Dynamic Analysis Integration:** Future pipelines could combine static analysis with fuzzing or symbolic execution to capture activation conditions and simulate exploit paths [45].
2) **Semantic Labeling:** Machine learning models could be trained on labeled malicious vs. benign functions to identify subtle intent-driven code structures [47].
3) **Behavioral Correlation:** Cross-chain event analysis (e.g., abrupt withdrawal patterns, metadata changes) could correlate flagged contracts with actual rug pull events.
4) **Cross-Ecosystem Risk Mapping:** Expanding analyses to other chains (Polygon, BNB Chain, Avalanche) may reveal whether these backdoor patterns are chain-specific or part of a broader trend.

The rapid rise of NFTs has brought immense innovation to digital ownership and asset representation but it has also created fertile ground for financial abuse via smart contract backdoors. In this study, we conducted one of the largest static analyses to date on 49,940 verified NFT smart contracts deployed on Ethereum, focusing on hidden logic patterns that facilitate rug pull attacks.

By leveraging Slither and designing a custom heuristic scoring system, we systematically identified and classified high-risk constructs such as selfdestruct, delegatecall, unrestricted mint() functions, and owner-only withdrawals. Our results reveal that more than 22% of NFT contracts exhibit multiple overlapping vulnerabilities, many of which are unlikely to exist unintentionally. These patterns strongly suggest a recurring design model in which control and withdrawal privileges are deliberately retained by the deployer, in direct contradiction to decentralization principles.

Through our risk classification, co-occurrence analysis, and contract-level case examples, we demonstrate that malicious contract logic is neither rare nor isolated it is embedded across a wide spectrum of deployed NFT projects. Moreover, our findings highlight the inadequacy of current audit practices and the need for stronger pre-deployment screening, automated risk assessment, and marketplace-level safeguards.

While our analysis is static and does not simulate live exploitability, it establishes a scalable and reproducible framework for flagging smart contract backdoors before they are abused in the wild. We hope this work contributes to a more secure, transparent, and accountable NFT ecosystem and that it catalyzes a broader shift toward embedding security as a first-class concern in smart contract development and deployment.

VIII. Conclusion

The rapid rise of NFTs has brought immense innovation to digital ownership and asset representation but it has also created fertile ground for financial abuse via smart contract backdoors. In this study, we conducted one of the largest static analyses to date on 49,940 verified NFT smart contracts deployed on Ethereum, focusing on hidden logic patterns that facilitate rug pull attacks [48].

By leveraging Slither and designing a custom heuristic scoring system, we systematically identified and classified high-risk constructs such as selfdestruct, delegatecall, unrestricted mint() functions, and owner-only withdrawal[49]. Our results reveal that more than 22% of NFT contracts exhibit multiple overlapping vulnerabilities, many of which are unlikely to exist unintentionally. These patterns strongly suggest a recurring design model in which control and withdrawal privileges are deliberately retained by the deployer, in direct contradiction to decentralization principles [50].

Through our risk classification, co-occurrence analysis, and contract-level case examples, we demonstrate that malicious contract logic is neither rare nor isolated, it is embedded across a wide spectrum of deployed NFT projects [51]. Moreover, our findings highlight the inadequacy of current audit practices and the need for stronger pre-deployment screening, automated risk assessment, and marketplace-level safeguards [52].

While our analysis is static and does not simulate live exploitability, it establishes a scalable and reproducible framework for flagging smart contract backdoors before they are abused in the wild. We hope this work contributes to a more

secure, transparent, and accountable NFT ecosystem and that it catalyzes a broader shift toward embedding security as a first-class concern in smart contract development and deployment [53].

References

[1] William Entriken and Dieter Shirley, et al. ERC-721: Non-Fungible Token Standard. Ethereum Improvement Proposals. https://eips.ethereum.org/EIPS/eip-721

[2] ERC-1155: Multi Token Standard. Ethereum Improvement Proposals. https://github.com/ethereum/eips/issues/1155

[3] Sharma Trishie and Agarwal Rachit, et al. "Understanding Rug Pulls: An In-depth Behavioral Analysis of Fraudulent NFT Creators" Association for Computing Machinery (2023).

[4] Escudero Daniel and Goyal Vipul, et al. "TurboPack: Honest Majority MPC with Constant Online Communication" Association for Computing Machinery (2022).

[5] Ma Fuchen and Ren Meng, et al. "Pied-Piper: Revealing the Backdoor Threats in Ethereum ERC Token Contracts" Association for Computing Machinery (2023).

[6] Daniel Perez and Benjamin Livshits, et al. "Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited" USENIX Association (2021).

[7] Haozhe Zhou and Amin Milani Fard, et al. "The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support" Journal of Cybersecurity and Privacy (2022).

[8] Feist Josselin and Grieco Gustavo, et al. "Slither: A Static Analysis Framework for Smart Contracts" 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB).

[9] Ye Jiaming and Ma Mingliang, et al. "Journal of Systems and Software" Elsevier BV (2022). https://arxiv.org/abs/1912.04466.

[10] "ERC-721 Non-Fungible Token Standard - Developer Documentation. Ethereum.org Developer Docs." https://ethereum.org/en/developers/docs/standards/tokens/erc-721/

[11] ERC-1155 Token Standard - Official Enjin Ecosystem Documentation. Enjin Platform. https://enjin.io/ecosystem/ethereum-foundation

[12] Hanting Chu and Pengcheng Zhang, et al. "A survey on smart contract vulnerabilities: Data sources, detection and repair" *Information and Software Technology.*

[13] Gerardo Iuliano and Dario Di Nucci. "Smart Contract Vulnerabilities, Tools, and Benchmarks: An Updated Systematic Literature Review" arXiv:2412.01719 (2025).

[14] Huynh, Phuong Duy and Dau, Son Hoang, et al. "Serial Scammers and Attack of the Clones: How Scammers Coordinate Multiple Rug Pulls on Decentralized Exchanges." Association for Computing Machinery (2025).

[15] Smart Contract Obfuscation Techniques. DeGatchi Technical Research. https://degatchi.com/articles/smart-contract-obfuscation/

[16] Malanii Oleh. "NFT Smart Contract Audit: Guide For Founders & Managers." *Hacken Security Research.* https://hacken.io/discover/security-audit-for-nft-guide-for-founders-and-managers/

[17] Fernando Richter Vidal and Naghmeh Ivaki, et al. "Vulnerability detection techniques for smart contracts: A systematic literature review." *Information and Software Technology.* https://www.sciencedirect.com/science/article/pii/S016412122400205X

[18] Cheng-Kang Chen and Wen-Yi Chu, et al. "Proxion: Uncovering Hidden Proxy Smart Contracts for Finding Collision Vulnerabilities in Ethereum." arXiv:2409.13563. https://arxiv.org/abs/2409.13563

[19] Top NFT Smart Contract Audit Tools in 2025: Ultimate Security Guide. Blockchain Security Analysis. https://blockchainpress.media/nft-smart-contract-audit-tools/

[20] Ghaleb, Asem and Pattabiraman, Karthik. "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection." Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis.

[21] ASSERT-KTH. (2024). DISL Dataset - 514,506 Unique Solidity Files. HuggingFace Datasets. https://huggingface.co/datasets/ASSERT-KTH/DISL

[22] Ethereum Foundation. (2024). Contracts - Solidity Documentation. Official Solidity Language Reference. https://docs.soliditylang.org/en/latest/contracts.html

[23] OpenZeppelin. (2024). OpenZeppelin Contracts - Library for Secure Smart Contract Development. GitHub Repository. https://github.com/OpenZeppelin/openzeppelin-contracts

[24] O'Reilly Media. (2024). Pragma, import, and comments - Ethereum Smart Contract Development. O'Reilly Learning Platform. https://www.oreilly.com/library/view/ethereum-smart-contract/9781788473040/1fe7a466-21a5-4726-8618-08de49730254.xhtml

[25] Solidity Documentation. (2024). Introduction to Smart Contracts. https://docs.soliditylang.org/en/v0.8.0/introduction-to-smart-contracts.html

[26] Slither Audited Smart Contracts Dataset. HuggingFace Datasets. https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts

[27] Lashkari, B., & Musilek, P. (2023). Evaluation of Smart Contract Vulnerability Analysis Tools: A Domain-Specific Perspective. Information, 14(10), 533. https://doi.org/10.3390/info14100533

[28] Ali F. Bakr and Khaled El Hagla, et al. "Heuristic approach for risk assessment modeling: EPCCM application (Engineer Procure Construct Contract Management)." https://doi.org/10.1016/j.aej.2012.09.001

[29] Kiani, R. and Sheng, V. S. (2024). Ethereum Smart Contract Vulnerability Detection and Machine Learning-Driven Solutions: A Systematic Literature Review. Electronics, 13(12), 2295. https://doi.org/10.3390/electronics13122295

[30] Jiachi Chen, Xin Xia, David Lo, and John Grundy. 2021. Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum. ACM Trans. Softw. Eng. Methodol. 31, 2, Article 30 (April 2022), 37 pages. https://doi.org/10.1145/3488245

[31] Halborn Security. (2023). Delegatecall Vulnerabilities In Solidity. Halborn Blog. Retrieved from https://www.halborn.com/blog/post/delegatecall-vulnerabilities-in-solidity

[32] OpenZeppelin. (2023). Admin Accounts and Multisigs. OpenZeppelin Blog. Retrieved from https://blog.openzeppelin.com/admin-accounts-and-multisigs

[33] Safe Global. (2024). Ethereum Smart Accounts. Retrieved from https://safe.global/

[34] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts SoK. In Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204.

[35] OpenZeppelin. (2024). The Parity Wallet Hack Explained. OpenZeppelin Blog. Retrieved from https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7

[36] Smart Contract Security Best Practices for NFT Creators. OpenSea Creator Documentation. Retrieved from https://docs.opensea.io/docs/smart-contract-security

[37] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18). Association for Computing Machinery, New York, NY, USA, 67–82.

[38] How to Secure NFTs: Part Two - NFT Smart Contract Security. Retrieved from https://www.certik.com/resources/blog/1Kl3XHKDdzIZvW0dyJLuSu-how-to-secure-nfts-part-two-nft-smart-contract-security

[39] Niosha Hejazi and Arash Habibi Lashkari. "A Comprehensive Survey of Smart Contracts Vulnerability Detection Tools: Techniques and Methodologies." https://doi.org/10.1016/j.jnca.2025.104142

[40] Ortu, M., Ibba, G., Destefanis, G. et al. Taxonomic insights into ethereum smart contracts by linking application categories to security vulnerabilities. Sci Rep 14, 23433 (2024). https://doi.org/10.1038/s41598-024-73454-0

[41] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. 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). Association for Computing Machinery, New York, NY, USA, 557–560. https://doi.org/10.1145/3395363.3404366

[42] Jintao Huang, Ningyu He, Kai Ma, Jiang Xiao, and Haoyu Wang. 2023. Miracle or Mirage? A Measurement Study of NFT Rug Pulls. Proc. ACM Meas. Anal. Comput. Syst. 7, 3, Article 51 (December 2023), 25 pages. https://doi.org/10.1145/3626782

[43] Cyberscope. (2024). NFT Smart Contract Audit. Retrieved from https://www.cyberscope.io/nft-smart-contract-audit
[44] Kiani, R.; Sheng, V.S. Automated Repair of Smart Contract Vulnerabilities: A Systematic Literature Review. Electronics 2024, 13, 3942. https://doi.org/10.3390/electronics13193942
[45] Trail of Bits. (2023). Fuzzing on-chain contracts with Echidna. Trail of Bits Blog. Retrieved from https://blog.trailofbits.com/2023/07/21/fuzzing-on-chain-contracts-with-echidna/
[46] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020). Association for Computing Machinery, New York, NY, USA, 415–427. https://doi.org/10.1145/3395363.3397385
[47] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309
[48] Kaspersky. (2023). What Are NFT Rug Pulls? How To Protect Yourself From NFT Fraud? Retrieved from https://www.kaspersky.com/resource-center/preemptive-safety/nft-rug-pulls
[49] Y. Huang, Y. Bian, R. Li, J. L. Zhao and P. Shi, "Smart Contract Security: A Software Lifecycle Perspective," in IEEE Access, vol. 7, pp. 150184-150202, 2019, doi: 10.1109/ACCESS.2019.2946988
[50] Lamby, Metin and Zieglmeier, et al. "Trusting a Smart Contract Means Trusting its Owners: Understanding Centralization Risk." https://arxiv.org/abs/2312.06510v1.
[51] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18). Association for Computing Machinery, New York, NY, USA, 664–676. https://doi.org/10.1145/3274694.3274737
[52] Consensys Diligence. (2024). Smart Contract Audits. Retrieved from https://diligence.consensys.io/
[53] Amritraj Singh and Kelly Click, et al. "Sidechain technologies in blockchain networks: An examination and state-of-the-art review" https://doi.org/10.1016/j.jnca.2019.102471.
[54] "Red Teaming the Mind of the Machine: A Systematic Evaluation of Prompt Injection and Jailbreak Vulnerabilities in LLMs" arXiv:2505.04806. https://arxiv.org/abs/2505.04806
[55] Fei, J., Chen, X. and Zhao, X. (2023). MSmart: Smart Contract Vulnerability Analysis and Improved Strategies Based on Smartcheck. Applied Sciences, 13(3), 1733. https://doi.org/10.3390/app13031733
[56] "Securing Genomic Data Against Inference Attacks in Federated Learning Environments" arXiv:2505.07188. https://arxiv.org/abs/2505.07188